

Complexity and Correctness of a Super-Pipelined Processor



Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Jochen Preiß

preiss@de.ibm.com

Saarbrücken, April 2005

Tag des Kolloquiums: 29. April 2005
Dekan: Prof. Dr. Jörg Eschmeier

Vorsitzender des Prüfungsausschusses: Prof. Dr. Gerhard Weikum
Erstgutachter: Prof. Dr. Wolfgang J. Paul
Zweitgutachter: Priv. Doz. Dr. Silvia M. Müller
akademischer Beisitzer: Dr. Sven Beyer

*Out Out!!
You demons of stupidity!!
– Dogbert*

Danke

Diesen Abschnitt möchte ich all denen widmen, die zum Gelingen dieser Arbeit beigetragen haben.

Zuallererst möchte ich meinen Eltern dafür danken, dass sie mich in allen Phasen meiner Ausbildung unterstützt haben und mir gerade in schwierigeren Zeiten stets ein Rückhalt waren.

Mein ganz besonderer Dank gilt auch Herrn Prof. Paul für die Vergabe dieses interessanten und herausfordernden Themas und für die wissenschaftliche Unterstützung.

Bei Christian Jacobi möchte ich mich für das Korrekturlesen dieser Arbeit und die vielen Vorschläge, die mir geholfen haben, diese Arbeit immer weiter zu verbessern, bedanken.

Meinen Freunden von der Uni Werner Backes, Christoph Berg, Sven Beyer, Mark Hillebrand, Thomas In der Rieden, Michael Klein und Dirk Leinenbach möchte ich danken für die fruchtbaren Diskussionen und das gute Klima, gefördert durch Tischfußball, Skat- und Doppelkopf-Abende und das Verschonen meines Rasens.

Meinen Kollegen bei IBM, insbesondere Cédric Lichtenau und Thomas Pflüger, gilt mein Dank für ihr Verständnis und die Unterstützung gerade in der letzten Phase meiner Dissertation.

Abstract

This thesis introduces the $DLX_{\pi+}$, a super-pipelined processor with variable cycle time. The cycle time of the $DLX_{\pi+}$ may be as low as 9 gate delays (including 5 gate delays for registers), which is assumed to be a lower bound for the cycle time. For the parts of the $DLX_{\pi+}$ that significantly differ from previous implementations correctness proofs are provided. Formulas are developed which compute restrictions to the parameters of the $DLX_{\pi+}$, e.g., the maximum number of reservation station entries for a given cycle time. The formulas also compute what modifications to the base design have to be made in order to realize a certain cycle time and what the impact is on the number of pipeline stages. This lays the foundation for computing the time per instruction of the $DLX_{\pi+}$ for a given benchmark and different cycle times in future work in order to determine the “optimum” cycle time.

Kurzzusammenfassung

In dieser Arbeit wird die $DLX_{\pi+}$ eingeführt, ein super-gepipelineter Prozessor mit variabler Zykluszeit. Die Zykluszeit der $DLX_{\pi+}$ kann bis auf 9 Gatter-Delays (inklusive 5 Gatter-Delays für Register) reduziert werden, was als untere Schranke für die Zykluszeit angesehen wird. Für die Teile der $DLX_{\pi+}$, die sich signifikant von bisherigen Implementierungen unterscheiden, werden Korrektheits-Beweise geliefert. Desweiteren werden Formeln entwickelt, die Beschränkungen für die Parameter der $DLX_{\pi+}$ wie zum Beispiel die maximale Anzahl von Reservation Station Einträgen für eine gegebene Zykluszeit berechnen. Die Formeln errechnen ausserdem welche Modifikationen am Basis-Design notwendig sind, um eine bestimmte Zykluszeit zu erreichen und welchen Einfluss dies auf die Anzahl der Pipeline-Stufen hat. Damit wird die Grundlage gelegt, um als zukünftige Arbeit die benötigte Zeit pro Instruktion der $DLX_{\pi+}$ für einen gegebenen Benchmark bei verschiedenen Zykluszeiten zu berechnen und damit die “optimale” Zykluszeit zu bestimmen.

Extended Abstract

In order to increase the performance of a processor regarding a specific benchmark one can either decrease the cycle time of the processor or the CPI (cycles per instruction) that the processor needs for the benchmark. Usual ways to decrease the CPI are, e.g., pipelining, out-of-order execution, branch prediction, or super-scalar designs. This thesis focuses on the cycle time.

The cycle time of a processor can be improved by increasing the number of pipeline stages of the processor and therefore decreasing the amount of work to be done in each stage. This is called super-pipelining. Note that super-pipelining may increase the CPI for several reasons. Due to the increased number of pipeline stages, data dependencies may have a larger impact. Also, the fewer amount of logic that fits into one cycle may have a negative impact on the micro-architecture, e.g., reduce the maximum number of possible reservation stations entries. This may increase the frequency of stalls and therefore increase the CPI. Thus, the minimum cycle time may not be the optimal cycle time for a design and a given benchmark.

This thesis introduces the $DLX_{\pi+}$, a super-pipelined processor with variable cycle time, i.e., with a variable number of pipeline stages. For computation of cycle time and cost of the $DLX_{\pi+}$ the technology independent gate model from [MP00] is used. The cycle time of the $DLX_{\pi+}$ may be as low as 9 gate delays (including 5 gate delays for the registers). For comparison, a 16 bit addition (which has 12 combinational gate delays in the used model) needs less than half a cycle in the deeply pipelined Pentium 4 processor [HSU⁺01], but needs 3 cycles in the $DLX_{\pi+}$ with 9 gate delays cycle time.

The variant of the $DLX_{\pi+}$ with 9 gate delays cycle time is more a proof of concept rather than it is assumed to have a good performance. Therefore, the main part of this thesis only handles cycle times of at least 10 in order to simplify the design. A variant of the $DLX_{\pi+}$ with a cycle time smaller than 9 is not assumed to be possible, although no formal proof for this is provided.

In this thesis formulas are developed which compute restrictions to the parameters of the $DLX_{\pi+}$, e.g., the maximum number of reservation station entries for a given cycle time. Other formulas compute what modification to the base design have to be made in order to realize a certain cycle time and what the impact is on the number of pipeline stages. This lays the foundation to write a cycle-accurate $DLX_{\pi+}$ simulator, that computes the performance of the $DLX_{\pi+}$ for a given benchmark and different cycle times in future work. Using this simulator the optimum cycle time of the $DLX_{\pi+}$ for the benchmark could be determined.

The $DLX_{\pi+}$ is an out-of-order processor that uses the Tomasulo scheduler [Tom67]. The design is based on the work of Kröning [Krö99]. The instruction set architecture (ISA) is taken from the MIPS R3000 processor [KH92] with small modifications simplifying the adaptation of the design. This allows a simulation of the $DLX_{\pi+}$ with MIPS R3000 instruction traces [Hil95] of the SPEC92 benchmark [SPEC].

In order to realize the small cycle times, parts of the $DLX_{\pi+}$ differ significantly from the design presented by Kröning. In particular new stalling and forwarding techniques are used. If these techniques are used it is for example not longer obvious that a RAM access returns the correct result. Therefore, correctness proofs are provided for the critical parts of the $DLX_{\pi+}$.

Zusammenfassung

Um die Leistung eines Prozessors bezüglich eines spezifischen Benchmarks zu verbessern, kann man entweder die Zykluszeit oder die CPI (benötigte Anzahl von Zyklen pro Instruktion) des Prozessors reduzieren. Bekannte Methoden die CPI zu reduzieren sind zum Beispiel Pipelining, Out-of-order Execution, Branch Prediction oder superskalare Designs. In dieser Arbeit geht es hingegen in erster Linie um die Reduzierung der Zykluszeit.

Die Zykluszeit eines Prozessors kann durch Erhöhen der Anzahl von Pipeline-Stufen des Prozessors und damit durch Reduzierung der Arbeit, die in jeder dieser Stufen verrichtet werden muss, verbessert werden. Dies wird Super-Pipelining genannt. Man beachte, dass Super-Pipelining die CPI aus verschiedenen Gründen erhöhen kann. Auf Grund der größeren Anzahl von Pipeline-Stufen könne Daten-Abhängigkeiten einen größeren Einfluß haben. Ausserdem kann die geringe Menge von Logik, die in einen Zyklus passt, negative Auswirkungen auf die Mikro-Architektur des Prozessors haben, indem zum Beispiel die maximal mögliche Anzahl von Reservation Stations Einträgen reduziert wird. Dies kann die Häufigkeit von Stalls und damit die CPI erhöhen. Daher muss die minimale Zykluszeit nicht notwendigerweise die optimale Zykluszeit für ein Design und einen gegebenen Benchmark sein.

In dieser Arbeit wird die $DLX_{\pi+}$ eingeführt, ein super-gepipelineter Prozessor mit variabler Zykluszeit, das heisst mit variabler Anzahl von Pipeline-Stufen. Zur Berechnung von Zykluszeit und Kosten der $DLX_{\pi+}$ wird das von der Technologie unabhängige Gatter Model aus [MP00] verwendet. Die Zykluszeit der $DLX_{\pi+}$ kann bis auf 9 Gatter-Delays (inklusive 5 Gatter-Delays für die Register) reduziert werden. Zum Vergleich, die Berechnung einer 16 bit Addition (die in dem benutzten Gatter Model 12 Gatter-Delays benötigt) braucht weniger als einen halben Takt im tief gepipelineten Pentium 4 Prozessor [HSU⁺01], aber braucht 3 Takte in der $DLX_{\pi+}$ mit 9 Gatter-Delays Zykluszeit.

Die Variante der $DLX_{\pi+}$ mit 9 Gatter-Delays Zykluszeit dient nur als Machbarkeits-Beweis. Es wird nicht erwartet, dass sie eine gute Leistung erreicht. Deshalb betrachtet der Hauptteil dieser Arbeit nur Zykluszeiten von mindestens 10 um das Design zu vereinfachen. Eine Variante der $DLX_{\pi+}$ mit einer Zykluszeit von weniger als 9 wird nicht als möglich erachtet, auch wenn kein formaler Beweis dafür gegeben wird.

In dieser Arbeit werden Formeln entwickelt, die Beschränkungen für die Parameter der $DLX_{\pi+}$ wie zum Beispiel die maximale Anzahl von Reservation Station Einträgen in Abhängigkeit von der Zykluszeit berechnen. Andere Formeln errechnen, welche Modifikationen am Basis-Design notwendig sind um eine bestimmte Zykluszeit zu erreichen und welchen Einfluss dies auf die Anzahl der Pipeline-Stufen hat. Damit wird die Grundlage gelegt, um als zukünftige Arbeit einen Zyklus-genauen $DLX_{\pi+}$ -Simulator zu schreiben, der die Leistung der $DLX_{\pi+}$ für einen gegebenen Benchmark und verschiedenen Zykluszeiten berechnet. Mit diesem Simulator wäre es möglich, die optimale Zykluszeit der $DLX_{\pi+}$ für den Benchmark zu bestimmen.

Die $DLX_{\pi+}$ ist ein out-of-order Prozessor der den Tomasulo Scheduler [Tom67] benutzt. Das Design basiert auf der Arbeit von Kröning [Krö99]. Der Instruktions-Satz wurde mit kleinen Änderungen, die die Anpassung des Designs erleichtern, vom MIPS R3000 Prozessor [KH92] übernommen. Dadurch ist es möglich, die $DLX_{\pi+}$ mit Hilfe

von MIPS R3000 Traces [Hil95] des SPEC92 Benchmarks [SPEC] zu simulieren.

Um die geringe Zykluszeit zu erreichen, müssen Teile der $DLX_{\pi+}$ gegenüber dem Design von Kröning signifikant verändert werden. Insbesondere müssen neue Techniken zum Stallen und Forwarden eingeführt werden. Durch den Einsatz dieser Techniken ist es zum Beispiel nicht mehr offensichtlich, dass ein RAM-Zugriff die korrekten Daten liefert. Deshalb werden für die kritischen Teile der $DLX_{\pi+}$ Korrektheits-Beweise geführt.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Outline | 3 |
| 2 | Basics | 5 |
| 2.1 | Notation | 5 |
| 2.2 | Cost and Delay Model | 6 |
| 2.3 | Basic Circuits | 7 |
| 2.4 | Encodings | 8 |
| 2.5 | Pipelining | 9 |
| 2.5.1 | Stages | 9 |
| 2.5.2 | Computation of Stall Signals | 11 |
| 2.5.3 | Optimization of the Stall Computation | 13 |
| 2.5.4 | Maximum Delay of Stall Inputs | 17 |
| 2.6 | Pipelining of RAM Blocks | 18 |
| 2.6.1 | Forwarding | 19 |
| 2.6.2 | Forwarding with Stalling | 21 |
| 2.6.3 | Pipelining of the Forwarding Circuits | 21 |
| 2.6.4 | Cost and Delay | 25 |
| 3 | Tomasulo Algorithm | 27 |
| 3.1 | Overview | 27 |
| 3.2 | Basic Data Structures | 28 |
| 3.2.1 | Functional Units | 28 |
| 3.2.2 | Register Files and Producer Tables | 28 |
| 3.2.3 | Reservation Stations | 29 |
| 3.2.4 | Common Data Bus | 29 |
| 3.2.5 | Reorder Buffer | 29 |
| 3.3 | Instruction Execution | 29 |
| 3.3.1 | Decode | 29 |
| 3.3.2 | Dispatch | 31 |
| 3.3.3 | Execute | 31 |
| 3.3.4 | Completion | 31 |
| 3.3.5 | Retire | 31 |

| | | |
|----------|--|-----------|
| 4 | Processor Core | 33 |
| 4.1 | Decode | 33 |
| 4.1.1 | Overview | 33 |
| 4.1.2 | Operands | 35 |
| 4.1.3 | Instruction Decoding Circuit | 36 |
| 4.1.4 | Operand Generation | 37 |
| 4.1.5 | Destination Computation | 38 |
| 4.1.6 | Instruction Issue | 39 |
| 4.1.7 | Stalling | 45 |
| 4.1.8 | Cost and Delay | 47 |
| 4.2 | Dispatch | 51 |
| 4.2.1 | Entries | 52 |
| 4.2.2 | Reservation Station Control | 55 |
| 4.2.3 | Pipelining | 58 |
| 4.3 | Functional Units | 63 |
| 4.4 | Completion | 64 |
| 4.4.1 | Arbiter | 64 |
| 4.4.2 | Pipelining | 66 |
| 4.4.3 | Cost and Delay | 68 |
| 4.5 | Retire | 70 |
| 4.5.1 | Overview | 70 |
| 4.5.2 | Tag Check | 72 |
| 4.5.3 | Interrupt Handling | 72 |
| 4.5.4 | Cost and Delay | 75 |
| 4.6 | Reorder Buffer Environment | 76 |
| 4.6.1 | Overview | 76 |
| 4.6.2 | Pipelining of the Retiring-Context | 78 |
| 4.6.3 | Forwarding | 79 |
| 4.6.4 | Implementation of Forwarding | 81 |
| 4.6.5 | Control | 82 |
| 4.6.6 | Correctness | 87 |
| 4.6.7 | Delay Optimizations | 90 |
| 4.6.8 | Cost and Delay | 93 |
| 4.7 | Register File Environment | 94 |
| 4.7.1 | Forwarding | 95 |
| 4.7.2 | General Purpose Register File | 98 |
| 4.7.3 | Floating Point Register File | 99 |
| 4.7.4 | Special Purpose Register File | 100 |
| 4.7.5 | Cost and Delay | 103 |
| 4.8 | Producer Table Environment | 104 |
| 4.8.1 | Forwarding | 104 |
| 4.8.2 | Cost and Delay | 107 |

| | | |
|----------|---|------------|
| 5 | Memory Unit | 109 |
| 5.1 | Overview | 109 |
| 5.2 | Overview of the Data Cache | 110 |
| 5.2.1 | Execution of Memory Accesses | 111 |
| 5.2.2 | Cache Core and Main Memory | 112 |
| 5.2.3 | Speculation | 113 |
| 5.3 | Hit Computation | 113 |
| 5.3.1 | Overview of the Hit Signal Computation | 114 |
| 5.3.2 | Local Hit Signals | 116 |
| 5.3.3 | Static Hit Signals | 117 |
| 5.3.4 | Global Hit Signals | 119 |
| 5.3.5 | Actions | 120 |
| 5.3.6 | Stall Computation | 122 |
| 5.3.7 | Cost and Delay | 122 |
| 5.4 | Cache Core | 126 |
| 5.5 | Update Queue | 127 |
| 5.5.1 | Entries | 128 |
| 5.5.2 | Control | 130 |
| 5.5.3 | Delay Optimizations | 133 |
| 5.5.4 | Optimized Completion for Store Instructions | 136 |
| 5.5.5 | Cost and Delay | 137 |
| 5.6 | Read Queue | 140 |
| 5.6.1 | Cost and Delay | 141 |
| 5.7 | Stall Computation | 143 |
| 5.8 | Cost and Delay | 143 |
| 6 | Instruction Fetch | 149 |
| 6.1 | Instruction Fetch Mechanism | 149 |
| 6.1.1 | Overview | 149 |
| 6.1.2 | Clocking of the Instruction Fetch | 150 |
| 6.1.3 | Branch Prediction | 150 |
| 6.2 | Instruction Fetch Unit | 151 |
| 6.2.1 | Overview | 151 |
| 6.2.2 | Instruction Cache | 153 |
| 6.2.3 | Computation of the Next Fetch-PC | 155 |
| 6.2.4 | Instruction Fetch Control | 158 |
| 6.2.5 | Cost and Delay | 159 |
| 6.3 | Instruction Fetch Queue | 159 |
| 6.3.1 | IFQ Entries | 160 |
| 6.3.2 | Control | 161 |
| 6.3.3 | Cost and Delay | 161 |
| 6.4 | Instruction Register Environment | 162 |
| 6.5 | Branch Checking Unit | 162 |
| 6.5.1 | Stall Computation | 165 |
| 6.5.2 | Cost and Delay | 165 |
| 6.6 | Processor Flush | 166 |

| | | |
|----------|---------------------------------------|------------|
| 7 | Discussion | 169 |
| 7.1 | Stage depths below 5 | 169 |
| 7.2 | Gate Model | 173 |
| 7.3 | Overall Cost and Delay | 174 |
| 7.4 | Related Work | 178 |
| 8 | Summary | 181 |
| 8.1 | Future Work | 181 |
| A | Instruction set architecture | 183 |
| A.1 | Instructions | 183 |
| A.2 | Encoding | 184 |
| B | Emulation of a MIPS R3000 | 189 |
| C | Additional Circuits | 191 |
| C.1 | Basic Circuits | 191 |
| C.1.1 | Design | 191 |
| C.1.2 | Cost and Delay | 191 |
| C.2 | Instruction Decode | 193 |
| C.2.1 | Decode | 194 |
| C.2.2 | Destination computation | 196 |
| D | Functional Units | 199 |
| D.1 | Integer ALU | 199 |
| D.2 | Integer Multiplicative Unit | 201 |
| D.3 | Floating Point Units | 203 |
| D.4 | Memory Unit | 205 |
| D.4.1 | Shift for Store | 205 |
| D.4.2 | Shift for Load | 207 |
| E | Cost and Delay | 209 |

Chapter 1

Introduction

Over the past fifty years the performance of microprocessors has dramatically increased. The advances in lithography allowed the building of constantly smaller transistors. This made the transistors faster and also increased the number of available transistors. A larger number of transistors made it possible to additionally increase the performance of the processor by implementing more advanced and complex designs.

The performance of a processor regarding a specific benchmark can be measured in TPI, the average time per instruction. The TPI can be computed as the product of the cycle time of the processor and the CPI (cycles per instruction). To increase the performance of a processor one can either decrease the cycle time or the CPI. Known techniques that may decrease the CPI are, e.g., pipelining, out-of-order execution, branch prediction, or super-scalar designs. This thesis focuses on improving the cycle time of the processor.

If technology improvements are neglected and the total work for processing an instruction is not changed, a lower cycle time can only be achieved by increasing the number of pipeline stages of the processor. Increasing the number of pipeline stages over the 5 stages of a simple pipelined processor, e.g., the MIPS R3000 [KH92], is called super-pipelining. However, extensive super-pipelining in order to minimize the cycle time does not necessarily maximize the TPI, since it may have a negative impact on the CPI.

An increased number of pipeline stages increases the number of cycles needed for “critical” loops, e.g. the execution of an ALU instruction and the forwarding of its result to the following instructions, or the resolving of a branch misprediction. Thus, the penalty for data dependencies or mispredicted branches becomes higher. This leads to stall conditions occurring more often which increases the CPI. Note that a constant part of the cycle time is consumed by the register delay. Only the remaining part is available for useful work. Splitting the cycle time in half therefore reduces the useful work by more than the half. The number of cycles needed for a computation can be more than doubled. Hence, the gain due to the lower cycle time may be lower than the loss due to the increased CPI.

Additionally, the decreased logic depth that fits into a cycle may have a negative impact on the micro-architecture of the processor. For example, if a register file access must be pipelined, forwarding of the write ports must be implemented which increases the combinational delay of the register file access. Also, it may happen that certain

parameters of the micro-architecture such as the number of reservation stations entries are bounded by the cycle time. Hence, for small cycle times it might be necessary to reduce the number of reservation station entries which could increase the CPI. Numerous other examples can be found throughout this thesis.

In order to investigate the side-effects and the limits of super-pipelining, this thesis introduces the $DLX_{\pi+}$, a super-pipelined processor with a variable cycle time. The cycle time and the cost of the $DLX_{\pi+}$ is computed using the technology independent gate model from [MP00]. Additionally to the cycle time the $DLX_{\pi+}$ supports other variable parameters, e.g., cache size, number of functional units, or number of reservation station entries.

The minimum cycle time of the $DLX_{\pi+}$ is only 9 gate delays (including 5 gate delays for the registers, thus leaving 4 gate delays for useful work). Note that the deeply pipelined Pentium 4 processor can compute a 16 bit addition (which has 12 combinational gate delays in our model) in less than half a cycle [HSU⁺01]. Hence, based on our model the amount of useful work that can be done in one cycle of the Pentium 4 is at least six times higher than the 4 gate delays in one cycle of the $DLX_{\pi+}$ with minimum cycle time. Even though the delay model may not be accurate as it neglects wire delay and fanout, the error is probably much less than a factor of six. The minimum cycle time of the $DLX_{\pi+}$ is therefore assumed to be far smaller than the cycle time of the Pentium 4 processor.

Some critical circuits of the $DLX_{\pi+}$ need two levels of multiplexers which together have 4 gate delays and hence use up all the useful work that can be done with minimum cycle time. Although no formal proof is provided, the author therefore assumes that it is not possible to build a $DLX_{\pi+}$ with a cycle time below 9 without sacrificing, e.g., a best-case CPI of 1. On the other hand, several trade-offs needed to be made in order to realize the $DLX_{\pi+}$ with 9 gate delays cycle time that can significantly increase the CPI for realistic benchmarks. Therefore for simplicity the main part of this thesis only treats cycle times of at least 10.

For cycle times of 10 and above this thesis develops formulas that define the behavior of the $DLX_{\pi+}$. Dependent on the cycle time these formulas compute the necessary modifications, the number of pipeline stages of the different parts of the design, and the cost of the processor. Additionally, formulas are developed that compute restrictions to the parameters of the $DLX_{\pi+}$ for a given cycle time. Using these formulas one can write a cycle-accurate $DLX_{\pi+}$ simulator that computes the TPI of the $DLX_{\pi+}$ for a given benchmark and different cycle times. Hence, one can determine the “optimum” cycle-time giving maximum overall performance for the benchmark. This part is future work.

The design of the $DLX_{\pi+}$ is based on Kröning’s out-of-order variant [Krö99] of the DLX [HP96] implementation by Müller and Paul [MP00]. The instruction set architecture (ISA) is taken from the MIPS R3000 processor [KH92] with small modifications simplifying the adaptation of the design. In contrary to the design of Müller and Paul, the $DLX_{\pi+}$ supports integer multiplications and divisions and does not use a delayed PC. The choice of the MIPS R3000 ISA allows for an accurate simulation of the $DLX_{\pi+}$ by using MIPS R3000 instruction traces [Hil95] of the SPEC92 benchmark [SPEC]. A complete listing of the $DLX_{\pi+}$ instruction set can be found in appendix A.

The DLX implementation by Kröning is used as the starting point for the $DLX_{\pi+}$. However, most circuits of Kröning's design have been redesigned. The changes either decreased the combinational delay of the design or were necessary to allow small cycle times. Especially an instruction fetch mechanism using branch prediction had to be introduced.

1.1 Outline

This thesis is structured as follows: the basics needed for the design of the $DLX_{\pi+}$ are presented in chapter 2. Chapter 3 describes the Tomasulo algorithm that is used by the $DLX_{\pi+}$ in order to execute instructions out-of-order. The design of the processor core of the $DLX_{\pi+}$ with a cycle time of 10 and above is presented in chapters 4 to 6. Chapter 4 details the core of the $DLX_{\pi+}$, chapters 5 and 6 detail the design of memory unit and instruction fetch. The results of this thesis including the modifications necessary for the $DLX_{\pi+}$ with a cycle time of 9 are discussed in chapter 7. A summary is given in chapter 8.

Chapter 2

Basics

In this chapter basic concepts used in this thesis are discussed. The notation and naming conventions are summarized in section 2.1. Section 2.2 introduces the gate model used to compute cost and delay of circuits. The basic circuits used in this thesis are presented in section 2.3. Section 2.4 introduces half-unary encoding used throughout this thesis. Sections 2.5 and 2.6 discuss pipelining of circuits and RAM blocks. These techniques are essential for the design of the $DLX_{\pi+}$ processor and the general discussion simplifies the description in the later chapters.

2.1 Notation

In this thesis the following naming conventions will be used for circuits and signals:

- Circuit names are written in **sans serif**.
- Register and signal names are written in *italic*.
- The output signal of a register *reg* is also denoted by *reg*. The data input signal is denoted by *reg'*.
- A bus *bus* with indexes from *j* to *i* is denoted by *bus*[*j* : *i*].
- Signals and busses can be combined to a multi-bus. A signal *sig* of a multi-bus *mbus* is denoted by *mbus.sig*. The whole multi-bus is denoted by *mbus.★*.
- The outputs of a circuit *Circ* are often combined to the multi-bus *Circ.★*.
- The concatenation of two signals *sig1* and *sig2* is denoted by {*sig1*, *sig2*}. If the signals belong to the same multi-bus *mbus* the notation *mbus*.{*sig1*, *sig2*} is used.
- If multiple signals are differentiated by an index (e.g. *sig₀* to *sig_n*), then *sig_★* denotes all signals of this kind.

For readability the usage of \star to denote all signals of a multi-bus or all indexes may be used imprecisely if it is clear from the context which signals are meant. If the context differentiates the signals *mbus.a* and *mbus.★*, then *mbus.★* means all signals of the multi-bus except the signal *mbus.a*.

2.2 Cost and Delay Model

To compare the performance and the cost (i.e., the area) of different processor designs a simple gate model based on the gate model in [MP95] is used. All gates and registers have constant delay. Fanout is not taken into account. The cost and delay of the gates are summarized in table 2.1. A register has a delay of 4 for the outputs and additionally a setup time of 1 for the inputs resulting in an overall delay of 5.

| | INV | NAND | NOR | AND | OR | MUX | XOR | XNOR | REG |
|-------|-----|------|-----|-----|----|-----|-----|------|-----|
| cost | 1 | 2 | 2 | 2 | 2 | 3 | 4 | 4 | 8 |
| delay | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 4+1 |

Table 2.1: Cost and delay of gates

The delay of a combinational path in a circuit is defined as the sum of the delays of the gates on the path. The delay of a signal is the maximum delay of all combinational paths from a register to the signal (excluding the delay of the register). The delay of a circuit is the delay of the longest combinational path from an input or a register inside the circuit to an output or a register inside the circuit. The following notations are used for cost and delay of gates, signals, and circuits:

- For a gate $GATE$ or a register REG , the delays are denoted by D_{GATE} and D_{REG} . The cost are denoted by C_{GATE} and C_{REG} .
- The delay of the signal sig is denoted by $D(sig)$.
- $D(sig_1, sig_2)$ denotes the maximum delay of two signals sig_1 and sig_2 .
- For signals sig_1 and sig_2 , $D(sig_1 \rightsquigarrow sig_2)$ denotes the maximum delay of all combinational paths from sig_1 to sig_2 .
- For a circuit $Circ$, $D(Circ)$ denotes the delay and $C(Circ)$ denotes the cost of the circuit.






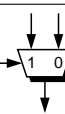


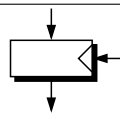
| INV | NAND | NOR | AND | OR | MUX | XOR | XNOR | REG |
|---|---|---|---|---|---|--|---|---|
|  |  |  |  |  |  |  |  |  |

Table 2.2: Gate symbols

The symbols used for gates in this thesis are shown in table 2.2. Inverted lines are indicated by small circles at the input or output of gates or circuits (see e.g. the NOR gate). All registers have a clock enable signal. The clock enable signal is connected to the triangle shape of the register symbol. If no signal is connected to the triangle shape, the clock enable is tied to one, i.e., the register is always clocked.

Registers are assumed to deliver both the negated and the non-negated value. Thus, it is possible to replace any AND or OR gate on the critical path by NAND and NOR gates using de Morgan's law. For the same reason all inverters on the critical path

can be removed. If a signal is used in multiple critical paths, it may be necessary to compute both the negated and non-negated value. For the sake of readability in the designs AND- and OR-gates will be used, but with the reduced delay to reflect a design with NAND- and NOR-gates. The revised delay of inverters, AND- and OR-gates is summarized in table 2.3.

| | INV | AND | OR |
|---|-----|-----|----|
| D | 0 | 1 | 1 |

Table 2.3: Revised delay of inverters, AND-, and OR-gates

A RAM block with A lines, D data bits, and a single read/write port is denoted by $\text{RAM}(A, D)$. Cost and delay of such a RAM block can be computed by the following formula from [MP95]:

$$C(\text{RAM}(A, D)) = 3 \cdot (A + 3) \cdot (D + \lceil \log \log D \rceil),$$

$$D(\text{RAM}(A, D)) = \begin{cases} \lceil \log D \rceil + \lceil A/4 \rceil & A \leq 64 \\ 3 \cdot \lceil \log A \rceil + 10 & A > 64 \end{cases}.$$

RAM blocks may have multiple read and write ports. The write ports are numbered from 1 to w . If multiple write accesses have the same target address, the write ports with smaller index have higher priority.

A RAM block with r read ports and w write port is denoted by $\text{RAM}(A, D, r, w)$. Cost and delay of this RAM block is based on the delay of a simple RAM block. The formula is taken from [Krö99]:

$$C(\text{RAM}(A, D, r, w)) = C(\text{RAM}(A, D)) \cdot (0.4 + 0.3 \cdot (r + 2w)),$$

$$D(\text{RAM}(A, D, r, w)) = D(\text{RAM}(A, D)) \cdot (0.5 + 0.25 \cdot (r + 2w)).$$

At higher frequencies it is not possible to access a RAM block in a single cycle. A RAM block which needs c cycles for every access is denoted by $\text{RAM}(A, D, r, w, c)$. The additional registers increases the cost of the RAM block by 10% per cycle.

$$C(\text{RAM}(A, D, r, w, c)) = C(\text{RAM}(A, D, r, w)) \cdot (0.9 + 0.1 \cdot c).$$

The design of pipelined RAM blocks that take multiple cycles for accesses is detailed in section 2.6. The given cost does not include any additional circuits needed for forwarding between the write and the read ports (see section 2.6.1).

2.3 Basic Circuits

Basic circuits such as adder, decoder, etc. which are used in this thesis are not discussed in detail. For cost and delay of the basic circuits and the design of a half-unary find-last-one circuit see appendix C.1. The symbols for the basic circuits are shown in table 2.4






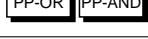




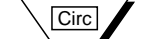
| | |
|---|--|
|  | Decoder / Encoder |
|  | Select Circuit (multiplexer with unary select signals) |
|  | Find-First-One Circuit / Find-Last-One Circuit |
|  | Half-unary Find-Last-One Circuit |
|  | Adder / Incrementer |
|  | Parallel-Prefix-OR / Parallel-Prefix-AND |
|  | Left-Shifter / Right-Shifter |
|  | Cyclic-Left-Shifter / Cyclic-Right-Shifter |
|  | Equality Checker / Test against constant k |
|  | AND-Tree / OR-Tree |
|  | Tree of associative circuit Circ |

Table 2.4: Basic Circuits

2.4 Encodings

The binary encoding with n bits of a number i with $0 \leq i < 2^n$ is denoted by $(i)_{bin(n)}$. If the width of the encoding is clear from the context, it can be omitted, i.e., the encoding can be denoted by $(i)_{bin}$. The number represented by an binary encoding v of length n is denoted by $\langle v \rangle$. Thus:

$$\langle v \rangle = \sum_{j=0}^{n-1} v[j] \cdot 2^j.$$

In multiple parts of the design of the $DLX_{\pi+}$ unary respectively half-unary encodings are used to represent numbers. The unary or half-unary encoding of length n of a number i is denoted by $(i)_{un(n)}$ respectively $(i)_{hun}$. It is defined by:

$$\begin{aligned} (i)_{un(n)} &:= 0^{n-i-2}, 1, 0^i, \\ (i)_{hun(n)} &:= 0^{n-i-2}, 1^{i+1}. \end{aligned}$$

The value represented by a vector v in unary or half-unary encoding is denoted by $\langle v \rangle_{un}$ respectively $\langle v \rangle_{hun}$. Thus, assuming v is a valid encoding it holds:

$$\begin{aligned} \langle v \rangle_{un} &= j \text{ if } v[j] = 1, \\ \langle v \rangle_{hun} &= \max\{j | v[j] = 1\}. \end{aligned}$$

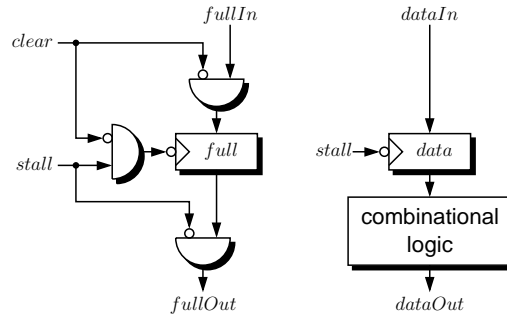


Figure 2.1: Stage

For unary and half-unary encodings, incrementers and decrementers can be implemented by one bit shifters, which have a constant delay of D_{MUX} :

$$\begin{aligned} (i+1)_{un} &= (i)_{un}[n-2:0], 0, & (i-1)_{un} &= 0, (i)_{un}[n-1:1], \\ (i+1)_{hun} &= (i)_{un}[n-2:0], 1, & (i-1)_{hun} &= 0, (i)_{un}[n-1:1]. \end{aligned}$$

Unary and half-unary encodings additionally allows the comparison of the represented value against a given constant with zero delay since this information can be directly derived from the signal with the corresponding index. Unary encodings are usually used if it must be checked whether the value is equal to the constant, half-unary encodings are used if it must be checked whether the value is larger than a constant. For a vector v in unary respectively half-unary encoding and a number j holds:

$$\begin{aligned} (v)_{un} &= j \text{ iff } v[j] = 1, \\ (v)_{hun} &\geq j \text{ iff } v[j] = 1 \end{aligned}$$

Note that in half-unary encoding the bit 0 is always one. Thus, it can often be removed to reduced the size of the vector. In this case the value 0 is represented by all bits being zero.

2.5 Pipelining

2.5.1 Stages

The circuits of the processor are divided into *stages*. A stage is a combinational circuit with a set of input registers (see figure 2.1). The delay of the combinational circuit is called *combinational delay* of the stage. Usually a stage has an explicit *full* register indicating whether the stage contains valid information. A stage is called *full* if the full bit is set. The clear signal *clear* resets the full bit, thus invalidating the content of the stage. The stall signal *stall* is active if the registers of the stage may not be updated.

A sequence of numbered stages, where the outputs of stage i are used as inputs of stage $i+1$ is called *pipeline* [Kog81] (see figure 2.2). If the signal name sig is used in multiple stages, the index of the stage is added to the signal name (sig^i) to distinguish the signals. The combinational delay of a pipeline is the sum of the combinational delays of the stages. Usually all stages of a pipeline have a common clear signal.

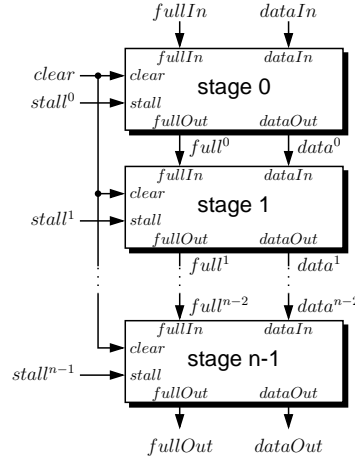


Figure 2.2: Pipeline with n stages

If the clear signal is not active, the flow of information through the pipeline is steered by the stall signals. Assume the stage i is full. If the stage $i + 1$ is stalled (i.e., the signal $stall^{i+1}$ is active) the information in stage i cannot proceed to the next stage. Then the stage i must also be stalled because otherwise the information in stage i would be overwritten. If the stage $i + 1$ is not stalled, but the stage i is stalled (e.g., due to a cache miss), the output full bit of stage i which is the input full bit of stage $i + 1$ must be invalidated. Otherwise the information of stage i would be duplicated. If a stage is not full it does not have to be stalled, as no information could be overwritten or duplicated.

The stall engine of [Krö01] also computes additional update enable signals that control the update of the data registers. The update enable signal of a stage i is activated if the signal $stall^i$ is not active and the full bit of the stage $i - 1$ is set. Hence, the stage i is only updated if valid information flow from stage $i - 1$ to i . However, this is not necessary for correctness, since the content of the data register may be arbitrary if the full bit is not set. Therefore, the update enable signals are omitted in this thesis.

In some circuits, parts of the registers of a stage have to be updated even if the stage is stalled. These registers are not directly controlled by the stall signal, but the new value of the registers often depends on the stall signal. Note that the delay of the stall signal may be large and may increase combinational delay of the stage.

In order to combine multiple pipelines, a pipeline has two additional stall signals $stallIn$ and $stallOut$. The input signal $stallIn$ must be active if the pipeline may not output data on its data output $dataOut$. The output signal $stallOut$ is active if the first stage of the pipeline cannot accept new data on its data input $dataIn$, i.e., $stallOut = stall^0$.

Consider a stage with a combinational delay D . The input registers have a delay of $D_{REG} = 4$. Due to the setup time of registers (which is 1), the stage bounds the cycle time τ to be at least $D + 5$. To allow cycle times smaller than $D + 5$, the stage can be replaced by a pipeline of multiple stages that computes the same outputs. This is done by splitting the combinational circuit in parts and adding registers which store the intermediate results, called “pipelining the stage”.

If a certain cycle time τ has to be reached, pipelining of the circuit must be done

such that the combinational delay of each stage may be at most $\delta := \tau - 5$. This maximum value for combinational delay of the stages δ is called *stage depth*. In this thesis δ is considered instead of the cycle time τ to reflect the frequency of the processor.

The transformation of a circuit into a pipeline of s stages changes the number of cycles needed to compute the result. For many circuits presented in this thesis the value of s is not relevant for the correctness of the processor. For example, it does not matter if a floating point computation is divided into 2 or 5 stages. It can be chosen such that the pipeline adheres to the maximum stage depth δ . In this case this thesis only describes the combinational circuit.

The transformation of a stage into a pipeline with c stages increases the cost of the circuit (mainly) by the cost of the staging registers. Computing the exact number of staging registers is usually difficult and needs to be done for each s separately, because it largely depends on the width of the intermediate results. In this thesis the additional cost is only approximated by:

$$(c - 1) \cdot \lceil (I + O)/2 \rceil \cdot C_{REG}. \quad (2.1)$$

where I is the number of inputs and O is the number of outputs of the combinational circuit. This includes all additional hardware of the pipelining including the stall computation and the buffer circuits (see the following section).

2.5.2 Computation of Stall Signals

A stage i of a pipeline can be stalled for two reasons. The stage i can generate the stall itself, e.g., a cache stage might generate a stall due to a detected cache miss. This is indicated by the signal $genStall^i$. If the stage $i + 1$ is stalled, the stage i has also to be stalled as the information in stage i cannot proceed to the stage $i + 1$. Both cases can be ignored, if stage i is not full ($full^i = 0$). In this case the registers of the stage do not contain valid information and the stage can therefore receive new data. To summarize, the stall signal of a stage is computed as:

$$stall^i = full^i \wedge (genStall^i \vee stall^{i+1}).$$

Similar to pipelines two signals $stallIn^i$ and $stallOut^i$ are defined for every stage i . The signal $stallIn^i$ corresponds to the signal $stall^{i+1}$, $stallOut^i$ corresponds to $stall^i$.

A pipeline stage including the computation of the stall signal is shown in figure 2.3. The dashed line can be ignored for now. The full output $fullOut$ is only set to zero if the signal $genStall$ is active in contrary to figure 2.1 where the stall signal is used to reset $fullOut$. This simplification can be made as the information in the stage cannot be duplicated if $genStall$ is not active. Otherwise the stage can only be stalled if the signal $stallIn$ is active. In this case the succeeding stage is stalled too, will thus not be updated, and hence the succeeding stage ignores the full output.

Consider a pipeline with c stages. Assume that no stage can generate a stall, i.e.,

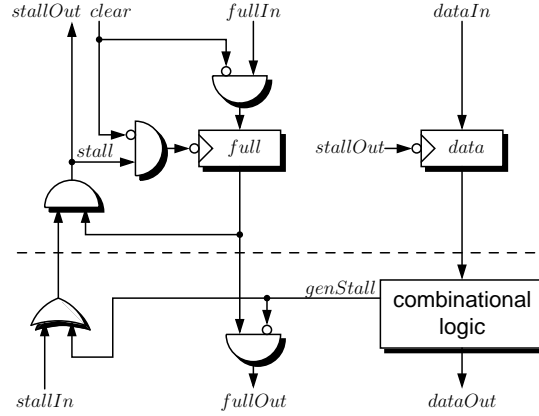


Figure 2.3: A stage with stall computation

$genStall^i = 0$ for all i . This simplifies the computation of the stall signal to:

$$\begin{aligned} stall^i &= full^i \wedge stall^{i+1} \\ &= \bigwedge_{j=i}^{c-1} full^j \wedge stallIn \end{aligned}$$

Thus, a stage can only be stalled if all succeeding stages are full. Assume stage i is the non-full stage with the highest index. If the stall input of the pipeline is active all stages with index higher than i are stalled and all stages with index lower or equal to i are not stalled. This removes the invalid information in stage i (called pipeline-bubble removal).

Theorem 2.1. *If the above implementation of the stall computation is used, the combinatorial depth D of a pipeline may be at most:*

$$D \leq \delta \cdot 2^\delta$$

Proof. Let c be the number of stages of the pipeline. It must hold $c \geq D/\delta$. The stall signal for stage 0 is computed as AND of the full bits of all stages and the input bit. Thus, the delay is at least $D_{AND} \cdot \lceil \log(c+1) \rceil$. The stall signal must be computed in one cycle. Thus:

$$\begin{aligned} \delta &\geq D_{AND} \lceil \log(c+1) \rceil \geq \log(c) \geq \log(D/\delta) \\ \Leftrightarrow 2^\delta &\geq D/\delta \\ \Leftrightarrow \delta \cdot 2^\delta &\geq D \end{aligned}$$

□

The combinational delay of the multiplicative floating point unit used in this thesis is 168. The theorem bounds the stage depth δ to be larger than 5 since $5 \cdot 2^5 = 160$. Hence, in order to reduce the stage depth to 5 or below, a different implementation for stall computation must be found.

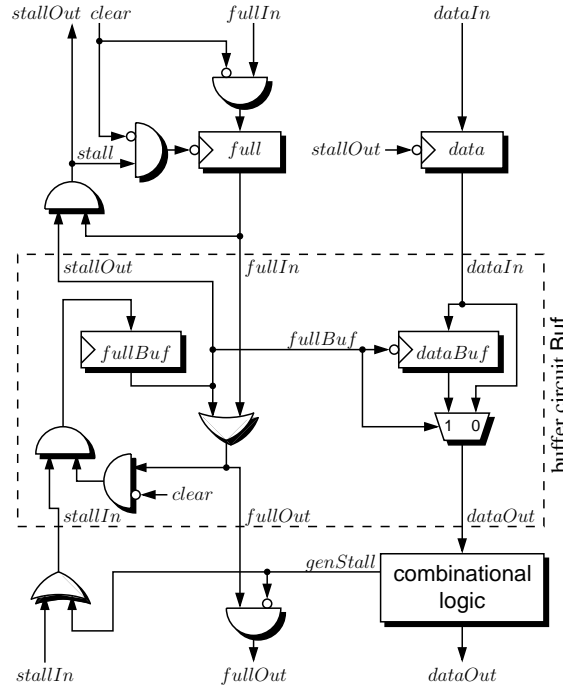


Figure 2.4: Stage with buffer circuit

2.5.3 Optimization of the Stall Computation

The simplest way to raise the bound given by the theorem is to change the computation of the stall signal such that pipeline-bubbles are no longer removed. For correctness it is usually sufficient to compute the stall signal the following way:

$$stall^i = genStall^i \vee stall^{i+1}.$$

Yet the minimum stage depth is still bounded by the delay of the OR of all signals $genStall^*$ and the delay of the stall input of the pipeline. The delay of the stall input can be significant, e.g., if the information in the last stage in the pipeline can flow into multiple succeeding pipeline (as during decode where an instruction can be issued to different reservation station (see chapter 3)). Then the stall input must be computed from the stall output of all acceding pipelines.

The bound given by the signals $genStall^*$ and the stall input is highly implementation dependent and therefore not treated in detail. Instead a more sophisticated solution to reduce the bound for the stage depth is described. This solution reduces the delay of the stall signals by pipelining the stall computation itself, i.e., registers are inserted in the stall computation circuit. This can be done by inserting a buffer circuit between the registers of a stage and the combinational circuit, as shown in figure 2.4. The buffer circuit is inserted at the dashed line of figure 2.3.

As long as the stage is not stalled, the buffer circuit is transparent, i.e. the outputs of the buffer circuit are equal to the corresponding inputs. The signal $fullBuf$ is then 0 and hence connects the data output with the data input. However, if the stage is stalled, the content of the full and data registers is saved in the buffer circuit. This enables the stage to receive data from the preceding stage into the input registers without loosing

the current information; the stall needs not to be propagated to the preceding stage. If the buffer circuit contains a valid instruction, the output stall signal is active and the stage is stalled. Hence, the stage is stalled not earlier than one cycle after the input stall signal becomes active. As soon as the stage is no longer stalled, the saved information from the buffer circuit is sent, i.e., the buffer circuit is emptied, before the buffer circuit goes back into transparent mode.

The buffer circuit decreases the delay of the signal *stallOut* to the delay of an AND gate independent of the delay of the stall input *stallIn*. It divides the pipeline in two pipelines with smaller stall circuits. Hence, the maximum combinational delay is not longer limited by the stall signals. Note that each buffer circuit increases the combinational delay of the pipeline by the delay of a mux. The correctness of the buffer circuit is summarized in the following theorem. Note that the theorem only handles the buffer circuit. Thus the signals used in the theorem as, e.g., *stallIn* and *stallOut* describe the inputs and output of the buffer circuit and not the stage.

Theorem 2.2. *Assume the inputs of the buffer circuit obey the following properties. The clear signal is active exactly in cycle 0:*

$$clear^{(t)} = \begin{cases} 1 & \text{if } t = 0 \\ 0 & \text{if } t > 0 \end{cases}. \quad (P0)$$

The signal stallIn is live:

$$\forall t > 0 \exists t' : t' > t \wedge stallIn^{(t')} = 0. \quad (P1)$$

The data of all instructions which enter the buffer circuit are distinguishable:

$$\begin{aligned} \forall t, t' > 0 : fullIn^{(t)} = fullIn^{(t')} = 1 \wedge stallOut^{(t)} = stallOut^{(t')} = 0 \\ \wedge dataIn^{(t)} = dataIn^{(t')} \Rightarrow t = t'. \end{aligned} \quad (P2)$$

Then the buffer circuit adheres to the following statements:

The buffer circuit is empty in cycle 1:

$$fullBuf^{(1)} = 0. \quad (S0)$$

The signal stallOut is live.

$$\forall t > 0 \exists t' : t' > t \wedge stallOut^{(t')} = 0. \quad (S1)$$

Every instruction which enters the circuit leaves the circuit exactly in the next possible cycle:

$$\begin{aligned} \forall t > 0 : fullIn^{(t)} = 1 \wedge stallOut^{(t)} = 0 \\ \Rightarrow \{t' | fullOut^{(t')} = 1 \wedge stallIn^{(t')} = 0 \wedge dataOut^{(t')} = dataIn^{(t)}\} \quad (S2) \\ = \{\min\{t' \geq t | StallIn^{(t')} = 0\}\} \end{aligned}$$

The ability to distinguish the instruction is needed for statement (S2). It can be reached by adding a unique index to every instruction.¹ Since this index has no influence on the behavior of the buffer circuit, it must not be implemented in hardware to reach correctness. It is merely a means to state the theorem.

¹It is a common trick to use a infinite set of tags for a completeness criterion. In can be proven later on that a finite set suffices for correctness as, e.g., in [BJK⁺03].

Proof. Statement (S0): The statement follows directly from the construction of the clear signal.

Statement (S1): The output stall signal may only be active if the input stall signal was active in the preceding cycle.

$$\begin{aligned} stallOut^{(t)} &= fullBuf^{(t)} \\ &= (fullBuf^{(t-1)} \vee full^{(t-1)}) \wedge stallIn^{(t-1)} \\ &\leq stallIn^{(t-1)} \end{aligned}$$

The statement follows from property (P1).

Statement (S2): The equivalence of the sets is proven in two steps.

“ \supseteq ” Let t be such that $fullIn^{(t)} = 1$ and $stallOut^{(t)} = 0$. It follows:

$$fullBuf^{(t)} = stallOut^{(t)} = 0. \quad (2.2)$$

Let t' be $\min\{t' \geq t \mid stallIn^{(t')} = 0\}$. The following two cases can be distinguished:

$t' = t$: It follows:

$$\begin{aligned} fullOut^{(t')} &= fullOut^{(t)} = fullIn^{(t)} \vee fullBuf^{(t)} \stackrel{(2.2)}{=} fullIn^{(t)} = 1, \\ dataOut^{(t')} &= dataOut^{(t)} = \begin{cases} dataBuf^{(t)} & \text{if } fullBuf^{(t)} = 1 \\ dataIn^{(t)} & \text{if } fullBuf^{(t)} = 0 \end{cases} \\ &\stackrel{(2.2)}{=} dataIn^{(t)}. \end{aligned}$$

$t' \neq t$: Hence, $stallIn^{(t)} = 1$. It follows:

$$\begin{aligned} fullBuf^{(t+1)} &= (fullBuf^{(t)} \vee fullIn^{(t)}) \wedge stallIn^{(t)} \geq fullIn^{(t)} = 1, \\ dataBuf^{(t+1)} &= \begin{cases} dataBuf^{(t)} & \text{if } fullBuf^{(t)} = 1 \\ dataIn^{(t)} & \text{if } fullBuf^{(t)} = 0 \end{cases} \\ &\stackrel{(2.2)}{=} dataIn^{(t)}. \end{aligned}$$

By definition of t' for all \bar{t} with $t < \bar{t} < t'$ holds $stallIn^{(\bar{t})} = 1$. For these \bar{t} the following can be proven by induction:

$$\begin{aligned} fullBuf^{(\bar{t}+1)} &= (fullBuf^{(\bar{t})} \vee full^{(\bar{t})}) \wedge stallIn^{(\bar{t})} \\ &\geq fullBuf^{(\bar{t})} \stackrel{(Ind.)}{=} 1, \end{aligned} \quad (2.3)$$

$$\begin{aligned} dataBuf^{(\bar{t}+1)} &= \begin{cases} dataBuf^{(\bar{t})} & \text{if } fullBuf^{(\bar{t})} = 1 \\ dataIn^{(\bar{t})} & \text{if } fullBuf^{(\bar{t})} = 0 \end{cases} \\ &= dataBuf^{(\bar{t})} \stackrel{(Ind.)}{=} dataIn^{(\bar{t})}. \end{aligned} \quad (2.4)$$

Thus, the content of the registers $fullBuf$ and $dataBuf$ does not change as long as $stallIn$ and $fullBuf$ are active. For the cycle t' it follows:

$$\begin{aligned} fullOut^{(t')} &= fullIn^{(t')} \vee fullBuf^{(t')} \stackrel{(2.3)}{=} 1, \\ dataOut^{(t')} &= \begin{cases} dataBuf^{(t')} & \text{if } fullBuf^{(t')} = 1 \\ dataIn^{(t')} & \text{if } fullBuf^{(t')} = 0 \end{cases} \\ &\stackrel{(2.3)}{=} dataBuf^{(t')} \stackrel{(2.4)}{=} dataIn^{(t)}. \end{aligned}$$

Thus, t' is in the set $\{t' | fullOut^{(t')} = 1 \wedge stallIn^{(t')} = 0 \wedge dataOut^{(t')} = dataIn^{(t)}\}$

“ \subseteq ” Let t' be such that $fullOut^{(t')} = 1$, $stallIn^{(t')} = 0$ and $dataOut^{(t')} = dataIn^{(t)}$. Let t'' be $\max\{t'' < t' | stallOut^{(t'')} = 0\}$. It follows:

$$fullBuf^{(t'')} = stallOut^{(t'')} = 0 \quad (2.5)$$

$t'' = t'$: It follows:

$$\begin{aligned} fullIn^{(t'')} &\stackrel{(2.5)}{=} fullIn^{(t'')} \vee fullBuf^{(t'')} = fullOut^{(t'')} \\ &= fullOut^{(t')} = 1 \\ dataOut^{(t')} &= dataOut^{(t'')} = \begin{cases} dataBuf^{(t'')} & \text{if } fullBuf^{(t'')} = 1 \\ dataIn^{(t'')} & \text{if } fullBuf^{(t'')} = 0 \end{cases} \\ &\stackrel{(2.5)}{=} dataIn^{(t'')} \end{aligned}$$

$t'' \neq t'$: Hence, $fullBuf^{(t')} = stallOut^{(t')} = 1$. It follows:

$$\begin{aligned} dataOut^{(t')} &= \begin{cases} dataBuf^{(t')} & \text{if } fullBuf^{(t')} = 1 \\ dataIn^{(t')} & \text{if } fullBuf^{(t')} = 0 \end{cases} \\ &= dataBuf^{(t')} \end{aligned}$$

By definition of t'' for all \bar{t} with $t'' < \bar{t} < t'$ holds $stallOut^{(\bar{t})} = 1$. For these \bar{t} the following can be proven by induction:

$$fullBuf^{(\bar{t})} = stallOut^{(\bar{t})} = 1 \quad (2.6)$$

$$\begin{aligned} stallIn^{(\bar{t})} &\geq stallIn^{(\bar{t})} \wedge (fullBuf^{(\bar{t})} \vee fullIn^{(\bar{t})}) \\ &= fullBuf^{(\bar{t}+1)} \stackrel{(2.6)}{=} 1 \end{aligned}$$

$$\begin{aligned} dataOut^{(t')} &\stackrel{(Ind.)}{=} dataBuf^{(\bar{t}+1)} = \begin{cases} dataBuf^{(\bar{t})} & \text{if } fullBuf^{(\bar{t})} = 1 \\ dataIn^{(\bar{t})} & \text{if } fullBuf^{(\bar{t})} = 0 \end{cases} \\ &\stackrel{(2.6)}{=} dataBuf^{(\bar{t})} \end{aligned} \quad (2.7)$$

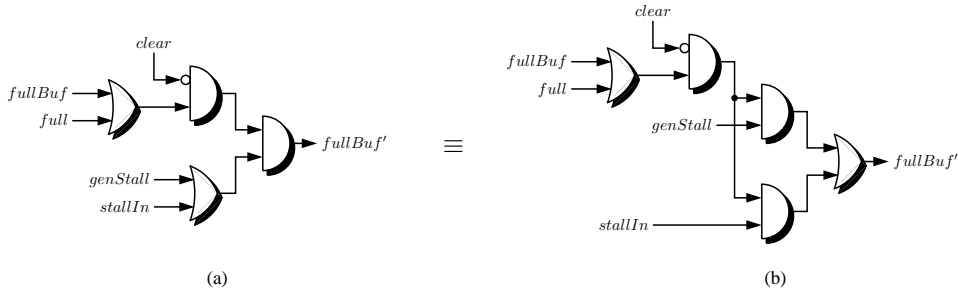


Figure 2.5: Optimized computation of $fullBuf'$

For the cycle t'' it follows:

$$\begin{aligned}
 fullBuf^{(t'')} &= stallOut^{(t'')} = 0 \\
 stallIn^{(t'')} &\geq stallIn^{(t'')} \wedge (fullBuf^{(t'')} \vee fullIn^{(t'')}) \\
 &= fullBuf^{(t''+1)} \stackrel{(2.6)}{=} 1 \\
 dataOut^{(t'')} &\stackrel{(2.7)}{=} dataBuf^{(t''+1)} = \begin{cases} dataBuf^{(t'')} & \text{if } fullBuf^{(t'')} = 1 \\ dataIn^{(t'')} & \text{if } fullBuf^{(t'')} = 0 \end{cases} \\
 &\stackrel{(2.8)}{=} dataIn^{(t'')}
 \end{aligned} \tag{2.8}$$

From the property (P2) it follows $t'' = t$. As $stallIn^{(\bar{t})} = 1$ for all $t \leq \bar{t} < t'$ it follows: $t' = \min\{t' \geq t \mid stallIn^{(t')} = 0\}$.

□

2.5.4 Maximum Delay of Stall Inputs

Let the stage i of a pipeline have a buffer circuit and assume none of the stages $j \in \{i+1, \dots, n\}$ for an $n \geq i$ has a buffer circuit. For all stages j the AND-gate that forces the clocking of the full register in case the clear signal is active (see figure 2.3) can be removed from the critical path by rebalancing. The computation of the stall signal for stage i comprises the stall signals for all stages j . Hence, the delay of the stall signals for all stages j is at most as high as the delay of the input of the register $fullBuf$ in the buffer circuit of stage i (see figure 2.4). Thus, if the delay of this signal $fullBuf^{i'}$ is at most δ the stall signals for all stages j can be computed in one cycle. Therefore, only the stages with a buffer circuits have to be checked whether the delay of the stall inputs is too large.

Figure 2.5(a) details the computation of the signal $fullBuf'$ from figure 2.4. The stall input $stallIn$ usually is computed by an AND-Tree. If the signal $genStall$ is not constantly zero the OR-gate hinders the integration of the last AND-gate into this tree. If the order of the gates is switched using the distributive law as shown in figure 2.5(b), the AND-gate can be integrated into the tree in order to reduce the delay.

Figure 2.6 depicts an example of the integration into the AND tree if the stall input is computed as AND of the signals $full^1$ to $full^k$. The overall delay of the circuit is equivalent to the delay of the AND-tree of the full bits with $1 + 2^{\lceil D_{OR}/D_{AND} \rceil}$ additional inputs (i.e., 3 additional inputs if the delay of AND and OR gates is equal as

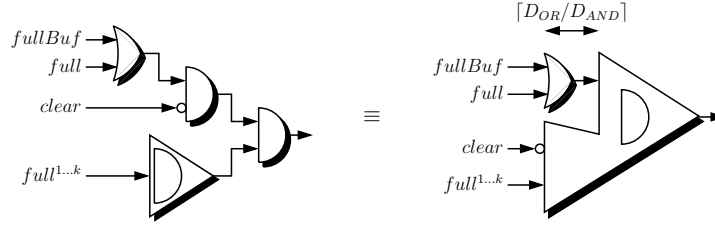


Figure 2.6: Merging logic into the tree of the stall input computation

in the gate model used in this thesis). Hence, if the input stall signal of a buffer circuit is computed by an AND-tree with k inputs and the stage with the buffer circuit cannot generate a stall, the signal $fullBuf'$ can be computed in one cycle if:

$$\delta \geq D(\text{AND-tree}(k + 3)).$$

If the stage with the buffer circuit can generate a stall the delay of the input of the register $fullBuf$ increases by the rightmost OR-gate in figure 2.5(b). Thus, the following equation must hold:

$$\delta \geq D(\text{AND-tree}(k + 3)) + D_{OR}.$$

Let i be chosen as above and n be the length of the pipeline. Then, the input of the register $fullBuf$ of stage i depends on the input stall signal $stallIn$ of the pipeline. If the stage i cannot generate a stall, it holds:

$$fullbuf^i = (fullBuf^i \vee full^i) \wedge \overline{clear} \wedge \bigwedge_{j=i+1}^n full^j \wedge stallIn$$

Thus, if the stall input $stallIn$ is computed by an AND-Tree with l inputs it must hold:

$$D(\text{AND-Tree}(n - i + 3 + l)) \leq \delta.$$

If $stallIn$ cannot be merged into the AND-tree of the full signals it must hold:

$$\begin{aligned} D(\text{AND-Tree}(n - i + 3)) &\leq \delta - D_{AND} & \text{and} \\ D(stallIn) &\leq \delta - D_{AND}. \end{aligned} \quad (2.9)$$

The delay of the stall computation increases by at least D_{OR} if any stage j for $i \leq j \leq n$ can generate a stall. Thus, in order to minimize the restrictions for the stall input $stallIn$, i should be chosen such that no such stage j can generate a stall. If last stage of a pipeline can generate a stall, i cannot be chosen as above. It must then hold:

$$D(stallIn) \leq \delta - (D_{OR} + D_{AND}). \quad (2.10)$$

2.6 Pipelining of RAM Blocks

In order to reduce the access time of a RAM block it is mandatory to also pipeline the RAM block. A schematic view of a RAM block with n address bits and m data

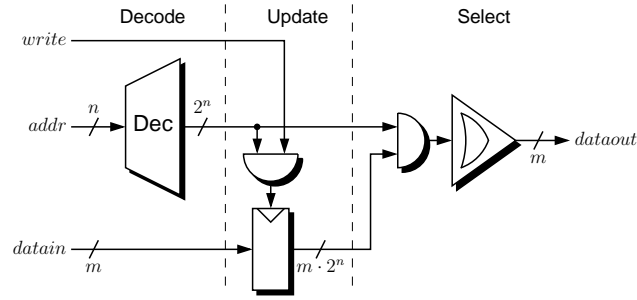


Figure 2.7: Schematic view of a RAM block

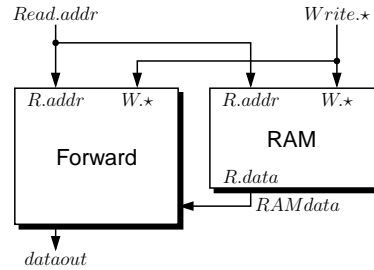


Figure 2.8: Forwarding of a write port

bits is shown in figure 2.7. A RAM block can be divided into three parts: the decode of the address bus, the update of the data registers and the selection of the addressed data [KP95].

Inserting registers in the decode and the select stage allows for smaller cycle times. However a read access to such a pipelined RAM takes into account only the write accesses that have been started before the read access. At the time the result of a read access is on the output data bus, the accessed address may already be overwritten by a succeeding write access. Thus, the RAM only returns the value of the accessed address at the time the read access entered the RAM. However in many applications in this thesis the read access must return the value of the accessed address at the time it leaves the RAM. To obtain the latest value of an address, all write accesses that have been started after the read access have to be forwarded to the output of that read.

2.6.1 Forwarding

If forwarding is used it could happen that an instruction does not enter or leave the RAM environment in the same cycle in which the instruction enters respectively leaves the RAM block. In the following the term “a RAM access is started” always means that the access enters the RAM environment (which is usually as soon as all signals needed for the access are available). “An access finishes” always means that the access leaves the RAM environment.

The forwarding of a write port W to a read port R is done using the forwarding circuit from figure 2.8. If the RAM block is pipelined into c stages, the forwarding circuit is also divided into c stages. The stages of the forwarding circuit contain the data corresponding to the read access in the corresponding stage of the RAM block.

In every cycle the forward circuit compares the newly started write access with all

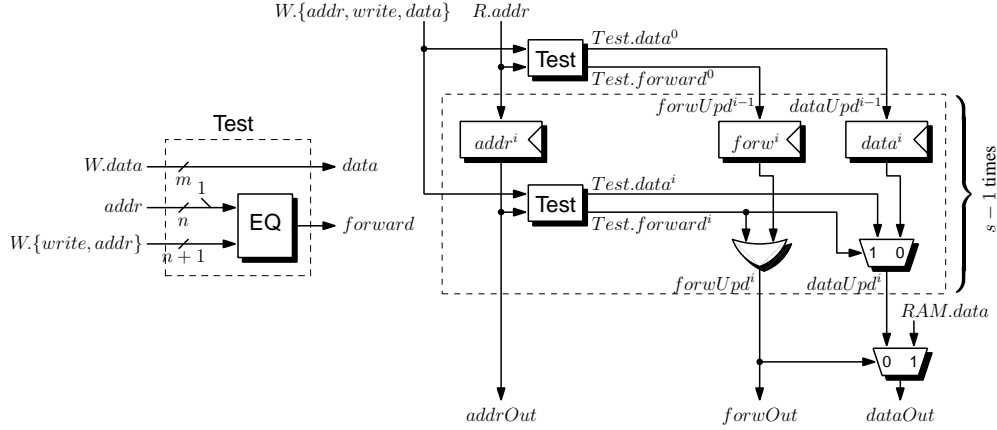


Figure 2.9: Forwarding circuit (with sub-circuit Test)

read accesses of all stages. If the write access overwrites the same address as a read access in the pipeline, the corresponding stage of the forwarding circuit is updated with the new write data. At the last stage of the forwarding circuit the read result is selected between the output of the RAM block and the data potentially saved in the forwarding circuit. In that way any write data to the read address is forwarded.

The details of the forwarding circuit are shown in figure 2.9. The address of the read access in stage i is saved in the register $addr^i$. The register $forw^i$ carries the information whether the stage i of the forwarding circuit holds valid forwarded write data. In that case the data of the last forwarded write access is saved in the register $data^i$. The stage i computes the updated values $forwUpd^i$ and $dataUpd^i$, which take the current write access on $W.*$ into account. The outputs of stage i are saved in the registers of stage $i + 1$.

A write access is forwarded to the read access in stage i , if the write signal is active and the address of the accesses are equal. This is indicated by the signal $Test.forward^i$ computed by the sub-circuit **Test** shown in the left part of figure 2.9. For simplicity the circuit also bypasses the data of the write access to the output $Test.data^i$. Using the signals $Test.forward^i$ and $Test.data^i$, the updated values $forwUpd^i$ and $dataUpd^i$ can be computed as:

$$forwUpd^i = forw^i \vee Test.forward^i,$$

$$dataUpd^i = \begin{cases} Test.data^i & \text{if } Test.forward^i = 1 \\ data^i & \text{if } Test.forward^i = 0 \end{cases}.$$

At the last stage $c - 1$ of the forwarding circuit, the signal $forwUpd^{c-1}$ is active, if there has been a write access started after the read access, which has overwritten the content of the accessed address. In this case the signal $dataUpd^{c-1}$ contains the newest content of the RAM address. If $forwUpd^{c-1}$ is not active, the output of the RAM $RAM.data$ contains the correct value. Hence, the current content of the accessed address can be computed as:

$$dataOut = \begin{cases} dataUpd^{c-1} & \text{if } forwUpd^{c-1} = 1 \\ RAM.data & \text{if } forwUpd^{c-1} = 0 \end{cases}.$$

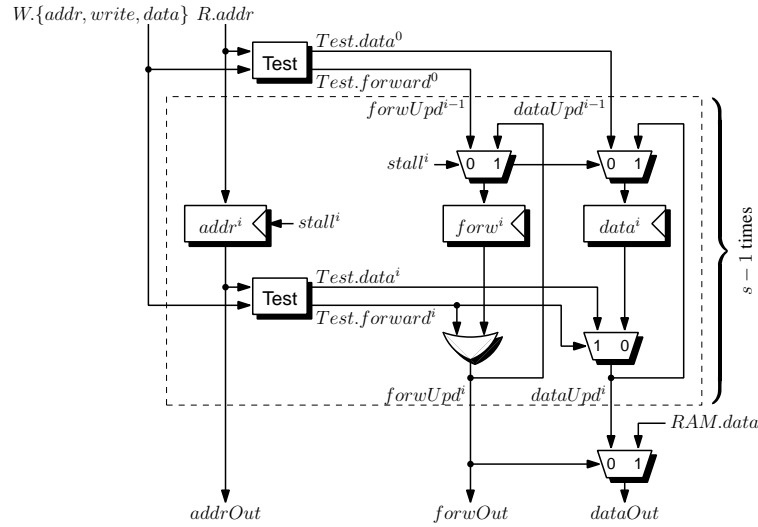


Figure 2.10: Forwarding circuit with stalling

2.6.2 Forwarding with Stalling

If the read access to a RAM block can be stalled it is not sufficient to update the read access with new write data only in those cycles the read advances to the next stage. Since new write accesses may be started even if the read access is stalled, forwarding must be possible within a stage when the read access does not progress.

Figure 2.10 shows a forwarding circuit with stalling. It is based on the forwarding circuit without stalling in 2.9. If the stage i is stalled, the updated values $forwUpd^i$ and $data^i$ must not be written into the registers of stage $i + 1$ but in the registers of stage i itself. Thus, a mux above the register $data$ and $forw$ of the stage i selects the outputs of the stages i and $i - 1$ as inputs for the registers depending on the stall signal $stall^i$:

$$data^{i'} = \begin{cases} dataUpd^i & \text{if } stall^i = 1 \\ dataUpd^{i-1} & \text{if } stall^i = 0 \end{cases},$$

$$forw^{i'} = \begin{cases} forwUpd^i & \text{if } stall^i = 1 \\ forwUpd^{i-1} & \text{if } stall^i = 0 \end{cases}.$$

Note that the stall signal is used in the combinational circuit to control the muxes above the registers. Thus, the stall signals of all stages i must adhere to:

$$D(stall^i) \leq \delta - D_{MUX}. \quad (2.11)$$

2.6.3 Pipelining of the Forwarding Circuits

Let n be the width of the address bus. The critical path of the circuit presented in section 2.6.1 goes from the write port $W.\star$ to the updated data $dataOut$. The delay of this path is

$$D(EQ(n+1)) + 2 \cdot D_{MUX}.$$

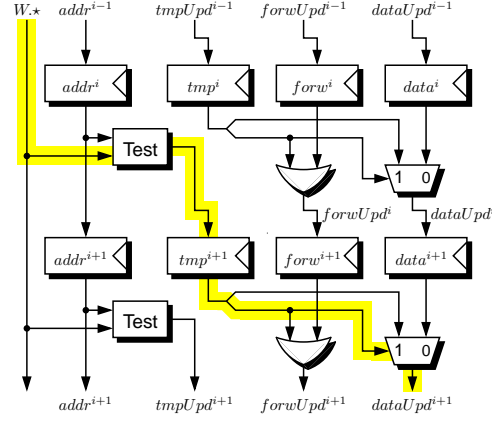


Figure 2.11: Pipelined forwarding circuit

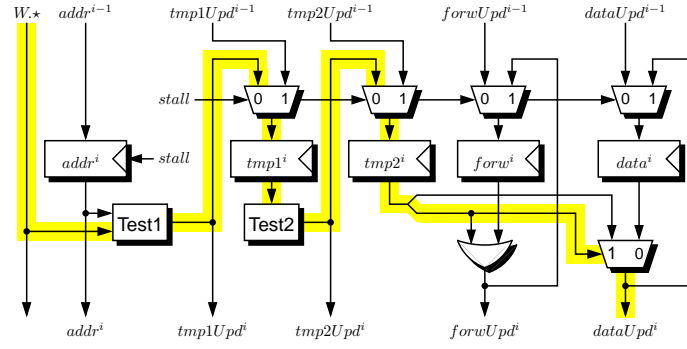


Figure 2.12: Pipelined forwarding circuit with stalling

To reduce the cycle time below this bound, the path from the write port to the updated data must be pipelined. The intermediate results of the pipelined computation are stored in registers and flow together with address, forward bit and data through the pipeline of the forwarding circuit. Figure 2.11 depicts as an example the stages i and $i + 1$ of a forwarding circuit without stalling where the computation of the updated data is split after the circuit **Test**. The register tmp^{i+1} is used to store the outputs of the circuit **Test** and to pipeline them into the next forwarding stage.

The forwarding circuit is now pipelined in two dimensions. Within the same cycle the read access moves to the next stage of the forwarding circuit and the computation of the updated data moves to the next part of the computation (see the path highlighted in figure 2.11). While the computation of the updated data for one write access moves to the second part, a new computation can be started for the next write access.

In the forwarding circuit with stalling the forwarding of the write port must proceed to the next part of the computation even if the instruction does not move to the next stage of the forwarding circuit. Therefore, above every register saving the temporary results a multiplexer is added. This multiplexer selects depending on the stall signal whether the register is updated with the output of the current or the preceding stage (analogously to the multiplexers above the forward bit and the data registers in figure 2.10).

Figure 2.12 depicts as an example a stage i of a forwarding circuit with stalling, where the circuit **Test** is divided into two circuits **Test1** and **Test2** and pipelining

registers are added after both circuits. The path highlighted in the figure shows the forwarding of the write port if the stage is stalled twice.

In order to pipeline the computation of the updated data in the forwarding circuit without stalling (called **Forward**), the computation must be divided into parts with a combinational delay of at most δ . In the forwarding circuit with stalling (called **ForwardStall**), every inserted pipelining register increases the combinational delay from the write input to the updated data by the delay of the multiplexer above the register. Hence, for the circuit **ForwardStall** the computation of the updated data must be divided into parts with a combinational delay of at most $\delta - D_{MUX}$.

Note that the new values written into the register *forw* and *data* in the circuit **ForwardStall** depend on the previous value of the registers. If a pipelining register would be added into this path, old data would be used to update the registers. For the data register this would mean that it holds the correct data only every other cycle. A general solution to pipeline these one-cycle dependencies cannot be given, but the resulting bound to the stage depth from this dependency is acceptable:

$$\delta \geq \max\{D_{OR}, D_{MUX}\} + D_{MUX} \quad (2.12)$$

In our gate model this only requires that $\delta \geq 4$.

Compensating the pipelining cycles

Assume k pipeline registers are inserted into each stage of the forwarding circuit. Then the forwarding circuit needs $k + 1$ cycles to forward the write port into the data registers. Thus, the write accesses which have been started in the last k cycles before the read access is finished are not taken into account for the read result. Due to the pipelining it is not possible to take all writes into account that have been started before a read access, but it often suffices to take all those writes into account, that have entered the RAM block at the time the read access finishes.

For a read access in order to take into account all the write accesses that have entered the RAM block at the time the read is finished, forwarding must be started at least k cycles before the write accesses enter the RAM block. This can be done by delaying the write port by k cycles and forwarding the un-delayed writes (see figure 2.13(a)).

Due to delaying of the write port, the up to k write accesses in the registers $W_{1,*}$ to $W_{k-1,*}$ have been started before the read access but not yet entered the RAM block. If the read access directly enters the RAM block when its started, these k writes are not taken into account for the result of the RAM block. Therefore, the read port is also delayed by k cycles. The forwarding circuit must be increased by k stages to align with the read access.

Delaying the write port usually has no impact on the performance (as long as the un-delayed write is forwarded). The delaying of the read port of the RAM increases the overall delay of the read access and therefore the overall delay of the circuit where the RAM is used.

Figure 2.13(b) shows a solution to take the k writes started directly before the read access into account for the result without delaying the read access. At the time the read is started, the addresses, write signals and data of these k write accesses are known and

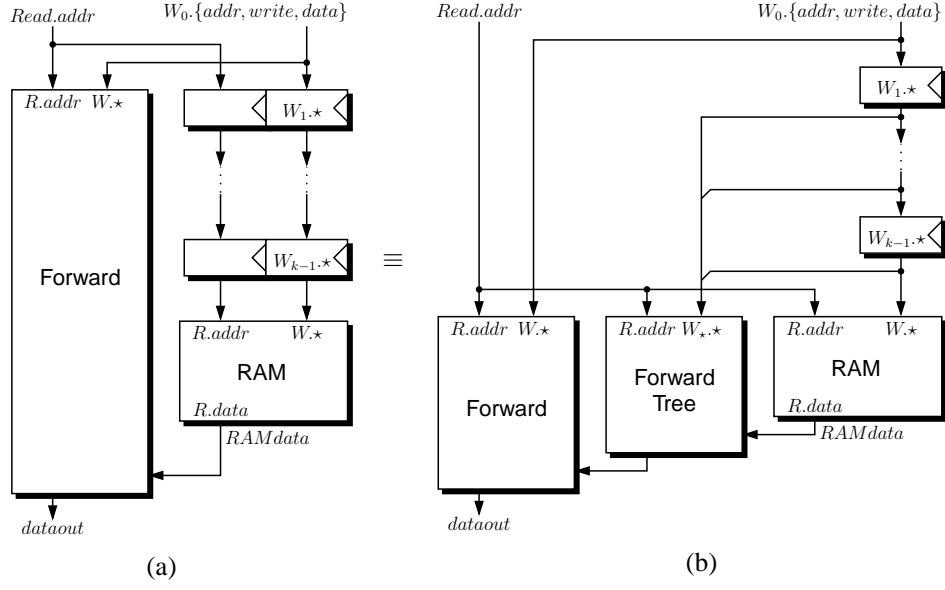


Figure 2.13: Forwarding with pipelined forwarding circuit

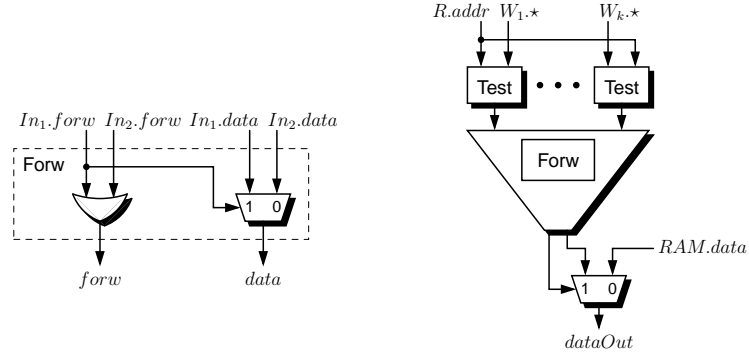


Figure 2.14: Forwarding Tree

stored in the registers $W_1.*$ to $W_{k-1}.*$. These write accesses are forwarded separately using a forwarding tree.

A circuit for a forwarding tree with k inputs is depicted in figure 2.14. All write accesses are tested in parallel using the circuit **Test**. The circuit **Forw** computes from the signals *forw* and *data* of two successive write accesses the combined values of the signals *forw* and *data*. The write access at input $In_1.*$ is assumed to be started after the write access at input $In_2.*$. Thus, the input $In_1.*$ has higher priority. If the forward bit $In_1.forw$ is active, the bus $In_1.data$ is the new data output, otherwise the bus $In_2.data$. The output *forw* is active if either of the inputs $In_*.forw$ is active. The circuits **Forw** can be arranged in a tree structure due to following lemma.

Lemma 2.3. *The function*

$$\circ : \mathbb{B}^2 \times \mathbb{B}^2 \mapsto \mathbb{B}^2, (f_1, d_1) \circ (f_2, d_2) \rightarrow (f_1 \vee f_2, f_1 d_1 \vee \overline{f_1} d_2)$$

is associative

Proof.

$$\begin{aligned}
((f_1, d_1) \circ (f_2, d_2)) \circ (f_3, d_3) &= (f_1 \vee f_2, f_1 d_1 \vee \overline{f_1} d_2) \circ (f_3, d_3) \\
&= (f_1 \vee f_2 \vee f_3, (f_1 \vee f_2)(f_1 d_1 \vee \overline{f_1} d_2) \vee \overline{(f_1 \vee f_2)} d_3) \\
&= (f_1 \vee f_2 \vee f_3, \\
&\quad f_1 f_1 d_1 \vee f_1 \overline{f_1} d_2 \vee f_2 f_1 d_1 \vee f_2 \overline{f_1} d_2 \vee \overline{f_1 f_2} d_3) \\
&= (f_1 \vee f_2 \vee f_3, f_1 d_1 \vee \overline{f_1} f_2 d_2 \vee \overline{f_1 f_2} d_3) \\
&= (f_1 \vee f_2 \vee f_3, f_1 d_1 \vee \overline{f_1} (f_2 d_2 \vee \overline{f_2} d_3)) \\
&= (f_1, d_1) \circ (f_2 \vee f_3, f_2 d_2 \vee \overline{f_2} d_3) \\
&= (f_1, d_1) \circ ((f_2, d_2) \circ (f_3, d_3))
\end{aligned}$$

□

Using the pipelined forwarding circuit it is possible to compute the content of a RAM block at the time a read access is returned even for a small stage depth δ . However the write accesses which have been started but have not yet entered the RAM block at the time the read access finishes cannot be forwarded. Additionally delaying the write port may have further implications to the circuit writing the RAM. Therefore, in the following sections for every RAM it is discussed which and how the write ports are forwarded to the read ports and why the forwarding suffices to guarantee the correctness.

2.6.4 Cost and Delay

If forwarding can be done without using a forwarding tree, the read access to the RAM is delayed by an additional mux for selecting between the forwarding data and the RAM output into the forwarding circuit. If a forwarding tree is used (it is assumed to be faster than the RAM access), the access is delayed by two muxes for selecting between the RAM output, the data output of the forwarding tree and the data output of the forwarding circuit (see figure 2.13(b)).

The longest combinational path for both forwarding circuits is the path from the inputs W_\star to the outputs $dataOut$. Let n be the number of address bits. Then the delay of the forwarding circuits are:

$$\begin{aligned}
D(\text{ForwardStall}(n)) &\leq D(\text{Test}(n)) + 2 \cdot D_{MUX}, \\
D(\text{Forward}(n)) &\leq D(\text{Test}(n)) + 2 \cdot D_{MUX}.
\end{aligned}$$

When pipelining the circuit **Forward** with n address bits the computation of the outputs from the inputs W_\star takes

$$c_F(n) \leq \left\lceil \frac{D(\text{Forward}(n))}{\delta} \right\rceil.$$

cycles. In order to pipeline the circuit **ForwardStall** an additional mux is needed before every inserted register. Thus, the path $W_\star \rightsquigarrow dataout$ must be divided into

parts with combinational delay of $\delta - D_{MUX}$. In the last stage no additional mux is needed. Thus, the computation of the outputs of the circuit takes

$$c_{FS}(n) \leq \left\lceil \frac{D(\text{Forward}(n)) - D_{MUX}}{\delta - D_{MUX}} \right\rceil$$

cycles (if it is not stalled).

Let n be the number of address bits, m the number of data bits, and c be the number of stages of the forwarding circuits. Without temporary registers the cost of the forwarding circuits is:

$$\begin{aligned} C(\text{Forward}(n, m, c)) &\leq C(\text{Test}) + m \cdot C_{MUX} + (c - 1) \cdot (C(\text{Test}) \\ &\quad + (m + n + 1) \cdot C_{REG} + C_{OR} + m \cdot C_{MUX}), \\ C(\text{ForwardStall}(n, m, c)) &\leq C(\text{Forward}(n, m, c)) + (c - 1) \cdot (m + 1) \cdot C_{MUX}. \end{aligned}$$

Let c_F and c_{FS} be the minimum number of cycles needed for forwarding respectively forwarding with stalling. The total cost of the forwarding circuits are (approximated with equation 2.1):

$$\begin{aligned} C(\text{Forward}(n, m, c, c_F)) &\leq C(\text{Forward}(n, m, c)) + (c - 1) \cdot c_F \\ &\quad \cdot \lceil ((m + n + n + 1) + (m + 1))/2 \rceil \cdot C_{REG} \\ &\leq C(\text{Forward}(n, m, c)) + (c - 1) \cdot c_F \\ &\quad \cdot (m + n + 1) \cdot C_{REG}. \end{aligned}$$

$$\begin{aligned} C(\text{ForwardStall}(n, m, c, c_{FS})) &\leq C(\text{ForwardStall}(n, m, c)) + (c - 1) \cdot c_{FS} \\ &\quad \cdot \lceil ((m + n + n + 1) + (m + 1))/2 \rceil \\ &\quad \cdot (C_{REG} + C_{MUX}) \\ &\leq C(\text{Forward}(n, m, c)) \\ &\quad + (c - 1) \cdot c_F \cdot (m + n + 1) \cdot (C_{REG} + C_{MUX}). \end{aligned}$$

The cost of the forwarding tree with k inputs (without registers) is:

$$C(\text{ForwardTree}(n, m, k)) \leq k \cdot C(\text{Test}) + (k - 1) \cdot C(\text{Forw}) + m \cdot C_{MUX}.$$

The forwarding tree needs to be divided into as many stages as the RAM access. Let c be the number of stages of the RAM access. Then the total cost of the pipelined forwarding tree is approximately:

$$\begin{aligned} C(\text{ForwardTree}(n, m, k, c)) &\leq C(\text{ForwardTree}(n, m, k)) + (c - 1) \\ &\quad \cdot \lceil ((k \cdot (m + n + n + 1)) + m + 1)/2 \rceil \cdot C_{REG} \\ &\leq C(\text{ForwardTree}(n, m, k)) \\ &\quad + (c - 1) \cdot k \cdot (n + m + 1) \cdot C_{REG}. \end{aligned}$$

Chapter 3

Tomasulo Algorithm

The DLX variant presented by Kröning which the $DLX_{\pi+}$ is based on, uses the Tomasulo algorithm [Tom67] to execute instructions out-of-order which allows for low CPI ratios [MLD⁺99]. It is assumed that the reader is familiar with this algorithm. Therefore this chapter gives only an informal description of the algorithm to define the terms used throughout this thesis. A formal description including correctness proofs can be found, e.g., in [KMP99].

The description of the Tomasulo algorithm is divided into three parts. In section 3.1 a general overview is given which defines the most important terms. Section 3.2 describes the basic data structures used by the algorithm. Finally in section 3.3 the algorithm is presented in more detail showing the execution of an example instruction.

3.1 Overview

For the Tomasulo algorithm every instruction which is being processed needs to be identified by a unique number. This number is called *tag*. The instructions which are processed at a given time are called *active instructions*.

Figure 3.1 shows an overview of the Tomasulo hardware. The instruction fetch unit does not differ from in-order processors. It loads the instruction stream from the main memory and delivers it in-order to the decode environment. In the decode environment the operands of the instructions are determined. If an operand is not computed yet, the decode environment determines the tag of the instruction which will compute the operand. Afterward the instruction is sent to a reservation station.

The instructions wait in the reservation stations until all operands are valid. The reservation stations check if the CDB carries the result of an instruction that is needed as operand for a waiting instruction. If this is the case the data is copied and the operand is validated. This forwarding from the CDB is called *snooping*. As soon as all operands are valid, the instruction is sent to a functional unit, independent of the instruction order.

The functional unit computes the result of the instruction and writes it to the common data bus. The common data bus forwards the result to the reservation stations and writes it in the reorder buffer. The reorder buffer reorders the instruction in program order before it writes the result in the register file.

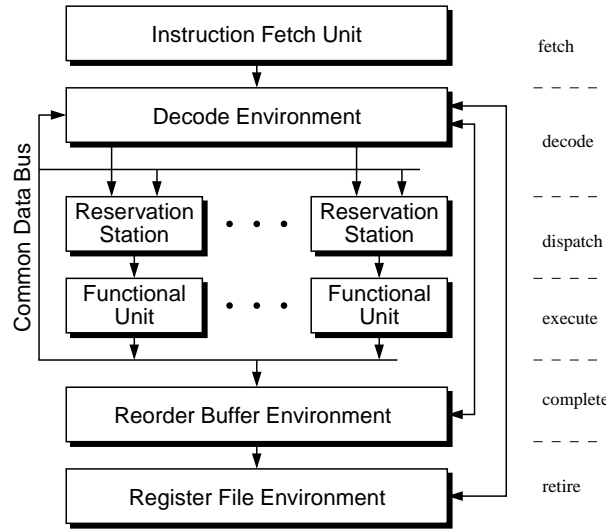


Figure 3.1: Overview of the Tomasulo hardware

| mnemonic | functional unit |
|----------|------------------------------------|
| BCU | branch checking unit |
| Mem | memory unit |
| IAlu | integer ALU |
| IMul | integer multiplicative unit |
| FAdd | floating point additive unit |
| FMul | floating point multiplicative unit |
| FMisc | floating point miscellaneous unit |

Table 3.1: Functional units types of the $DLX_{\pi+}$

3.2 Basic Data Structures

3.2.1 Functional Units

The *functional units* (FUs) perform the actual execution of the instructions. A processor can have different FU types, each executing only subsets of the instruction set. Multiple FUs of the same type are supported. The FU types of the $DLX_{\pi+}$ are listed in table 3.1. Tables A.1 to A.7 in the appendix list all $DLX_{\pi+}$ instructions sorted by the type of their functional unit.

3.2.2 Register Files and Producer Tables

The $DLX_{\pi+}$ has three different *register files* (RF): the general purpose register file (GPR), the floating point register file (FPR), and the special purpose register file (SPR). For every register file entry, the Tomasulo algorithm additionally stores a valid bit and a tag field. If no active instruction will write a register file entry, the content of the register file can be used as operand of a new instruction. In this case the valid bit of the register file entry is set to one. If one or more active instructions will write an entry, the valid bit is zero and the tag field stores the tag of the youngest active instruction which will write this entry.

The valid bits and the tag bits of the register file entries are saved in the *producer tables* (PT). Usually the producer tables have more access ports than the register files. Therefore, they are saved in different RAM blocks.

3.2.3 Reservation Stations

The *reservation stations* (RS) consist of multiple entries. Each entry can hold one instruction. The instructions wait in the entries until all operands are valid. To keep track of the state of the operands, each operand of an entry has a valid bit and a tag field, either indicating that the data in the entry is already valid or identifying the instruction which will eventually compute the value of the operand data.

The $DLX_{\pi+}$ has exactly one reservation station (of multiple entries) for every functional unit. Therefore, the reservation station is often identified with its functional unit. It is also possible to use only one reservation station per instruction type or even only one global reservation station [HP96]. For simplicity the latter cases are not treated in this thesis.

3.2.4 Common Data Bus

All functional units write the instruction results to the *common data bus* (CDB). To identify the current result on the CDB, the CDB has a valid and tag field. This relates the result to the instruction which computed that result. The common data bus writes the result to the reorder buffer. Also, the result is forwarded to the reservation stations. This allows the reservation stations to snoop on the CDB, i.e., check whether the result is needed as operand for a waiting instruction.

3.2.5 Reorder Buffer

The reordering of the instructions before the update of the register files is done by the *reorder buffer* (ROB). For every instruction, an entry in the reorder buffer is allocated in program order. For each entry the reorder buffer has a valid bit. This bit is active if the result of the instruction has already been computed. If the valid bit of the oldest instruction in the ROB is active, the instruction is removed from the ROB in order to write its result to the register file. The address of the reorder buffer entry of an instruction is used as tag for the instruction.

3.3 Instruction Execution

The execution of an instruction is done in six phases: fetch, decode, dispatch, execute, complete, and retire. The instruction fetch does not differ from in-order processors. Let I be an instruction which is being executed by the processor.

3.3.1 Decode

In the decode phase the opcode of the instruction I is decoded and the valid bit, the tag, and the data field of the operands are determined. The producer table entry of the destination register of I is updated and a new entry for I is reserved in the reorder

buffer. Afterward the instruction is sent to the reservation stations. This last step is called *issue*.

The decoding of the instruction I is straightforward and does not differ much from an in-order processor. The decode environment computes the addresses of the operand and the destination registers as well as the control signals used by the functional units.

For each operand the decode phase must determine the valid bit, the tag, and the data field. The correct value for the data field of an operand can be found at one of four different places:

- The register file: If no preceding active instruction writes the register file entry, the register file contains the valid data for the operand. In this case the valid bit of the corresponding producer table entry is set. If the valid bit is not set, the value tag_P of the tag field of the producer table entry identifies the latest instruction I_P which will compute the value of the operand.
- The reorder buffer: If the instruction I_P has already completed but is not retired yet, the data for the operand can be found in the reorder buffer. This can be tested by checking the valid bit of the reorder buffer entry of the instruction I_P . Note that the address of the entry of I_P in the reorder buffer is tag_P .
- The common data bus: If the instruction I_P is about to complete, the operand can be found on the CDB. This is the case if the valid bit of the CDB is active and the tag of the CDB equals tag_P .
- The reservation stations or the functional units: In this case the value of the operand is not computed yet.

In the first three cases the valid bit of the operand can be set to one and the data can be taken from the corresponding place. In the last case, the valid bit of the operand has to be set to zero and the tag field of the operand is set to tag_P . Then the instruction will wait in the reservation station until the result of the instruction I_P (identified by tag_P) becomes available on the CDB.

In parallel to the determination of the operands, the decode phase reserves a new entry in the ROB for the instruction I . For this the valid bit of the entry is reset. All the instruction's information which are known at decode time (e.g., the destination register address) are written into the corresponding ROB fields. Let tag be the address of the reorder buffer entry of I .

For the succeeding instructions to use the correct data, the producer table entry of the destination register of the instruction I is updated. The valid bit of the entry is set to zero and the tag is set to tag . This tells succeeding instruction that the new value of the register will be computed by instruction I .

At the end of the decode phase the instruction is sent to the reservation station of the functional unit that corresponds to the type of the instruction. If multiple functional units respectively reservation stations of one type exist, one of these reservation stations has to be chosen. In this thesis the first reservation station that is not full is used.

3.3.2 Dispatch

The instruction I waits in the reservation station until all of its operands are valid. Assume an operand of I is not valid and depends on instruction I_P . The Decode phase guarantees that in this case the instruction I_P is in a reservation station or a functional unit (see above). Hence, the result of I_P will eventually be sent via the common data bus. The reservation station entry of instruction I snoops on the CDB for the tag tag_P of the instruction I_P . If the tag of the CDB matches tag_P and the valid bit of the CDB is active, the data of the CDB is copied to the data field of the operand and the operand is marked as valid in the reservation station.

As soon as all operands are valid, the instruction I is sent to the functional unit (*dispatch*). If multiple instructions in a reservation station are valid at the same time, the oldest instruction is dispatched.

3.3.3 Execute

During the execute phase, the actual execution of the instruction is performed. This is done in the functional units. The functional units do not necessarily return the instructions in the order they enter. For example a floating point multiplication can overtake a floating point division [Jac02].

3.3.4 Completion

During the completion phase the results of the functional units are written to the ROB and forwarded to the reservation stations via the CDB. The number of functional units is usually larger than the number of results which fit on the CDB. Therefore, an arbiter decides which functional unit with an available result may write to the common data bus. During the completion phase the valid bit of the ROB entry of the instruction is set to one.

3.3.5 Retire

The original Tomasulo algorithm [Tom67] writes the register files out-of-order. For precise interrupt handling it is necessary to be able to restore the content of the register file as if the instructions would be executed in order. This can be very complex if instructions write the register file out-of-order. Therefore, the instructions are reordered into program order before updating the register files using the ROB [SP85].

To restore the program order, only the oldest instruction in the ROB is checked for whether it has already completed. In this case the instruction is taken out of the reorder buffer and the result is written into the destination register entry of the register files. The valid bit of the register file can then be set to flag valid data unless a younger active instruction writes to the same register.

In order to check whether a younger active instruction will write the same register, the producer table entry of the destination register is read. If the tag stored in the producer table matches the tag of the instruction I no younger instruction may write the same destination register. Otherwise it would have updated the tag of the producer table entry. In this case the valid bit of the producer table entry can be set.

In the retire phase the instruction is also checked whether it causes an interrupt or whether a branch misprediction occurred. In these cases all succeeding instructions are invalid and the processor has to be flushed. Since the register file is in a sequentially correct state, the producer table entries are all set to valid to indicate that no instructions will write the registers.

Chapter 4

Processor Core

In this chapter the design of the $DLX_{\pi+}$ core (without instruction fetch unit) is presented. The stage depth of the presented design is variable and is assumed to be at least 5 gate delays. Sections 4.1 to 4.5 describe the hardware for the five phases of the instruction processing in the core (decode, dispatch, execute, completion, and retire). Since the main RAM structures are accessed in multiple phases, they are described afterwards in the sections 4.6 to 4.8.

The design of the core is based on the Tomasulo DLX of Kröning [Krö99]. Splitting of the ROB into multiple smaller RAMs to reduce the number of ports was introduced by Hillebrand [Hil00]. Yet almost all non-trivial circuits had to be redesigned in order to maximize the performance and to allow a stage depth of 5 gate delays.

4.1 Decode

4.1.1 Overview

The decode phase is divided into the two sub-phases $D1$ and $D2$. In the sub-phase $D1$ the instruction word is decoded and the control signals for the instruction are computed. The valid bit, the tag, and the data of the operands are read from the current content of the producer tables respectively register files. In the sub-phase $D2$ the instruction is issued to a reservation station corresponding to the type of the instruction. In parallel the reorder buffer is checked if any of the instruction identified by the tags of the operands have already completed. If this is the case, the valid bit and the data field of the operand are updated. The decode phase uses the instruction register as input. The instruction register environment is described in section 6.4 as part of the the instruction fetch chapter.

Sub-phase $D1$

Figure 4.1 gives an overview of the sub-phase $D1$. The sub-circuit **Decode** computes the control signals for the current instruction. In parallel to the decoding the operands of the instruction are determined. For this the register files in the sub-circuit **RF** and the corresponding producer tables in the sub-circuit **PT** are accessed. Each register file type (GPR, FPR, and SPR) is accessed speculatively under the assumption that the operands are registers of this register file. The addresses of the accesses are computed

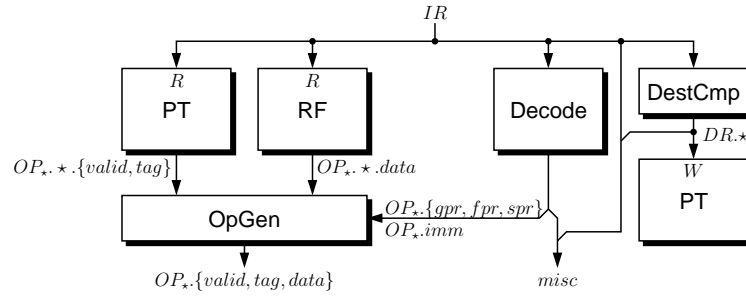


Figure 4.1: Decode sub-phase $D1^1$

inside the register file and producer table environments using the instruction register IR . The design of the register files and the producer tables are described in detail in sections 4.7 and 4.8.

The circuit **OpGen** selects for each operand i the register file which is used based on the outputs of the circuit **Decode**. If none of the register files is selected an immediate constant computed by **Decode** is used as operand data.

To update the destination register entry of the producer table, the register file and the address of the destination register must be known. These values are computed by the circuit **DestCmp**. Due to the delay of the circuit **DestCmp**, the update of the producer table entry of the destination register is started $c_{DC} \geq 1$ cycles after the read of the operand's producer table entries. The read access to the producer tables for the operands must return the content of the producer table after the decoding of all preceding instructions. Hence, to maintain correctness, the update of the destination register must be forwarded to the next c_{DC} instructions. This is done in the producer table environment (see section 4.8).

Apart from the instruction the instruction fetch unit delivers data to the decode phase which is needed for interrupts and branch checking, e.g., the PC of the instruction and the predicted target of branches. This data is not modified by the decode sub-phase $D1$.

Sub-phase $D2$

The design of the decode sub-phase $D2$ differs from the design proposed in [Krö99]. In Kröning's work the instructions first access the ROB and are then issued to the reservation stations. For the correctness of the Tomasulo algorithm it is necessary that no update of the ROB by the CDB is missed by the instruction until the instruction is written to a reservation station and starts to snoop on the CDB. This can be guaranteed easily if the whole decode phase including issuing of the instruction fits into one cycle as in the design presented by Kröning. However, for a small stage depth it is difficult to forward the CDB while issuing.

Figure 4.2 shows an overview of the decode sub-phase $D2$. The design presented in this thesis issues the instructions in parallel to the ROB access. This can be done because the operands of an instruction have consistent values for the valid bit, the

¹The producer table is accessed twice during decode. In order to emphasize that the two accesses are largely independent the producer table environment is depicted twice in the figure, even if it is implemented only once.

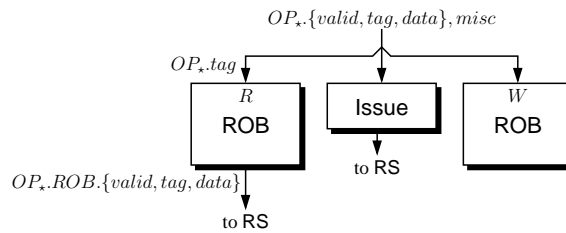


Figure 4.2: Decode sub-phase $D2^2$

tag and the data when the instruction leaves the decode sub-phase $D1$, i.e., if valid is active, the data field contains the correct operand data, otherwise the tag contains the tag of the instruction that computes the operand. If the result of the read access to the ROB by this instruction is available, it is used to update the instruction's operand data in the reservation station. For this the reservation stations snoop on the output of the ROB access in the same way they snoop on the CDB.

The ROB access and the issuing must not be aligned, i.e., if issuing is stalled, the ROB access can still progress. It must only be guaranteed, that the instruction is issued before the ROB access finishes. Otherwise the instruction would miss the result of the ROB access, as this only updates the reservation stations but not the instructions in the issue circuit. This may lead to a dead lock, since then the reservation station may wait indefinitely for receiving valid operand data.

The ROB read access by an instruction must take all writes to the ROB by the CDB into account, that have been started until the instruction is issued to a reservation station. This is guaranteed by the ROB environment (see section 4.6). Once an instruction is issued to a reservation station all updates to the ROB by the CDB are used to update the instruction's operands through snooping of the reservation station.

In parallel to the read access to the ROB a new ROB entry must be allocated for the instruction. This is done by resetting the valid bit of the ROB entry and writing all information about the instruction that is already known during decode into this entry (e.g., the destination address, the PC of the instruction, or interrupts occurring during fetch or decode). The design of the ROB environment is described in detail in section 4.6.

4.1.2 Operands

An instruction may have up to four operands Op_i for $i \in \{1, \dots, 4\}$. The first two operands have up to 64 bits for double precision operations. They are divided into a low part and $Op_i.lo$ and a high part $Op_i.hi$ each 32 bits wide. The operands Op_3 and Op_4 contain the floating point mask and the rounding mode which are needed by floating point instructions. The floating point mask is 5 bits wide, the rounding mode is 2 bits wide. These operands are not divided into high and low part. Note that the operand Op_3 and Op_4 always read the special register file.

In the Tomasulo algorithm each part of an operand consists of a *valid* bit, a *tag*, and a *data* field. The valid bit is set if the *data* field contains valid data. If the *valid* bit

²As in figure 4.1 the ROB environment is depicted twice to emphasize the two independent accesses to the ROB.

is not set, the *tag* field contains the tag of the instruction which computes this operand. In case an operand is not needed for an instruction, the valid bit is set to one in order to prevent the reservation station from waiting and snooping for this operand.

Each operand can be either an immediate constant or an entry of one of the register files **GPR**, **FPR**, and **SPR**. The signals for the operand Op_i from the register file are denoted by $Op_i.\mathcal{R}.\{valid, tag, data\}$. If the operand is an immediate constant the bus

$$Op_i.CO.\{valid, tag, data\} := \{1, 0^{l_{ROB}}, Op_i.imm\}$$

is used, where $Op_i.imm$ is the immediate constant for Op_i computed by the circuit **Decode**.

The memory unit **Mem** and the branch checking unit **BCU** may use a third operand as address offset. This operand is always an immediate constant, hence it is always valid and does not need to be updated in the reservation station. This operand may thus be treated like a control signal rather than as operand for the functional units. This saves the logic for this operand in the reservation station.

4.1.3 Instruction Decoding Circuit

The sub-circuit **Decode** of the decode sub-phase *D1* computes the following control signals for every instruction. The precise definition and the computation of the signal is described in appendix C.2.1:

- $FU.\{Mem, IAlu, IMul, FAdd, FMul, FMisc, BCU\}$: These signals indicate to which functional unit type an instruction is sent. If an instruction has been predicted to be a branch instruction (indicated by the signal $IFQ.pb$, see section 6.3), the instruction is sent to the BCU, independent of the real opcode (see the BCU section 6.5). To simplify the ROB environment, all instructions use an FU, even if they do not produce a result, e.g., due to an instruction page fault. Such instructions use the integer ALU as fake FU. This guarantees that exactly one of the signals $FU.\star$ is active at any cycle.
- $Op_i.\{gpr, fpr, spr\}$: These signals indicate from which register file the operand i is read ($i \in \{1, 2\}$). For each operand at most one of these signals may be active. If none of the signals is active, the operand is assumed to be an immediate constant.
- $Op_i.dbl$: For operands $i \in \{1, 2\}$, this signal is active if the operand is a double precision floating point register.
- $Op_i.imm$: If the operand $i \in \{1, 2\}$ is an immediate constant, the value of this constant is encoded in $OP_i.imm$. On an instruction memory interrupt the immediate constant is set to the PC of the instruction. This simplifies the reorder buffer (see section 4.5.3).
- *ill*: This signal is active if an illegal instruction occurred. This can happen, e.g., due to an invalid opcode or due to a double precision floating point access to an odd register address.

- *readIEEEf, writeIEEEf*: These signals are active if the instruction is a move instruction that reads respectively writes the special register *IEEEf*.
- *FPstore*: This signal is active for floating point stores.

Cost and Delay

Cost and delay of this circuit are (see appendix C.2.1):

$$\begin{aligned} C(\text{Decode}) &\leq 381 \cdot C_{AND} + 64 \cdot C_{OR} + 32 \cdot D_{MUX}, \\ D(\text{Decode}) &\leq 6 \cdot D_{AND} + 5 \cdot D_{OR}. \end{aligned}$$

4.1.4 Operand Generation

The circuit **OpGen** determines the operands of the instruction. If one of the control signals $OP_i.\{gpr, fpr, spr\}$ is active, the output for the operand i of the corresponding register file is selected. If none of the control signals is active, the immediate constant $OP_i.CO.\star$ is used.

The operands OP_3 and OP_4 are only used by floating point operations. They always read the same special registers: the floating point mask bits ($SPR[0]$) and the rounding mode ($SPR[6]$). Thus:

$$OP_i.\star := OP_i.SPR.\star \quad \text{for } i \in \{3, 4\}$$

The high part of the operands OP_1 and OP_2 is only used if the operand is a floating point double precision register (indicated by $OP_i.dbl = 1$). Thus, the high part always reads the floating point register file:

$$OP_i.hi.valid := \begin{cases} OP_i.FPR.hi.valid & \text{if } OP_i.dbl = 1 \\ 1 & \text{else} \end{cases} \quad \text{for } i \in \{1, 2\},$$

$$OP_i.hi.\{tag, data\} := OP_i.FPR.hi.\{tag, data\}.$$

The low part of the first two operands can be an immediate constant or a register from any register file. The signals $OP_i.\{gpr, fpr, spr\}$ from the circuit **Decode** determine which register file is used. At most one of these signals may be active. The low part of the first two operands can then be computed as:

$$OP_i.lo.\star := \begin{cases} OP_i.GPR.\star & \text{if } OP_i.gpr = 1 \\ OP_i.FPR.lo.\star & \text{if } OP_i.fpr = 1 \\ OP_i.SPR.\star & \text{if } OP_i.spr = 1 \\ OP_i.CO.\star & \text{else} \end{cases} \quad \text{for } i \in \{1, 2\}.$$

The reservation stations require four additional control signals $OP_i.depDbl$ and $OP_i.odd$ for $i \in \{1, 2\}$ to decide whether the data needed by an operand is on the low or the high part of an instruction's result. The signal $OP_i.depDbl$ is active if the operand i depends on a 64 bit result. The signal $OP_i.odd$ indicates, that the address of the operand OP_i is odd. Since 64 bit operands must read an even address it follows, that the operand is 32 bits wide and therefore only uses the low part of the operand. If

both signals are active, it follows, that the low part of the operand depends on the high part of an 64 bit result (64 bit instructions always write its low part in an even register and its high part in an odd register). In all other cases the low part of the operand depends on the low part of the result. The high part of an operand can always read the high part of a result as 32 bit instructions write their result on both the high and the low part of the CDB and the ROB (see section 4.3).

64 bit results are either written to the floating point register files or to the special purpose register file (by integer multiplications / divisions). The producer tables of these register files return the signals $OP_i.\mathfrak{R}.dbl$ indicating that an odd register is written by a 64 bit result for $i \in \{1, 2\}$ and $\mathfrak{R} \in \{FPR, SPR\}$. The overall signals $OP_i.depDbl$ and $OP_i.odd$ can be computed as:

$$OP_i.depDbl := \begin{cases} OP_i.FPR.depDbl & \text{if } OP_i.fpr = 1 \\ OP_i.SPR.depDbl & \text{if } OP_i.spr = 1 \\ 0 & \text{else} \end{cases}$$

The FPR and the SPR also return the lowest bit of the addresses used by the operands 1 and 2, $OP_i.\mathfrak{R}.addr[0]$. Then the signal $OP_i.odd$ can be computed as:

$$OP_i.odd := \begin{cases} OP_i.FPR.addr[0] & \text{if } OP_i.fpr = 1 \\ OP_i.SPR.addr[0] & \text{if } OP_i.spr = 1 \\ 0 & \text{else} \end{cases}$$

Cost and Delay

The circuit **OpGen** is implemented as a unary select circuit. It is controlled by the signals $OP_{\star}.\{gpr, fpr, spr\}$ and $(\overline{OP_{\star}.gpr} \wedge \overline{OP_{\star}.fpr} \wedge \overline{OP_{\star}.spr})$. The last signal can be computed by the circuit **Decode**, its delay is part of the delay of this circuit. Let l_{ROB} be the width of the tags. Cost and delay of the circuit **OpGen** are as follows:

$$\begin{aligned} C(\text{OpGen}) &\leq 2 \cdot C_{OR} + 2 \cdot 2 \cdot C_{AND} \\ &\quad + 2 \cdot (1 + 32 + l_{ROB}) \cdot C(\text{Sel}(4)) + 4 \cdot C(\text{Sel}(2)), \\ D(\text{OpGen}) &\leq D(\text{Sel}(4)). \end{aligned}$$

4.1.5 Destination Computation

The circuit **DestCmp** computes the signals $D.\star$ needed for updating the producer table entry of the destination register of the instruction. The valid bit of the producer table entry is always set to zero and the tag field is set to the tag of the instruction. The tag of the instruction is the current tail pointer of the ROB:

$$\begin{aligned} D.valid &:= 0, \\ D.tag &:= ROB.tail. \end{aligned}$$

For each register file \mathfrak{R} the circuit **DestCmp** computes a write signal $D.\mathfrak{R}.write$ and an address $D.\mathfrak{R}.addr$. It also computes a double signal for the register files FPR

and SPR (integer multiplications / divisions write their 64 bit result into two SPR register). As for the circuit **Decode** the detailed computation for this circuit is described in the appendix C.2.2.

At most one of the write signals $D.\mathcal{R}.write$ is active. This signal determines the register file that the result is written to. Thus, the actual address of the destination register $D.addr$ can be computed by selecting the addresses for the register files with the write signals. An instruction writes a double precision result if the floating point or the special purpose double bit is active. Thus:

$$D.addr := \begin{cases} D.GPR.addr & \text{if } D.GPR.write = 1 \\ D.FPR.addr & \text{if } D.FPR.write = 1 \\ D.SPR.addr & \text{if } D.FPR.write = 1 \\ \star & \text{else} \end{cases},$$

$$D.dbl := (D.FPR.dbl \wedge D.FPR.write) \vee (D.SPR.dbl \wedge D.SPR.write).$$

The signals $D.addr$, $D.dbl$, and $D.\mathcal{R}.write$ are saved in the ROB to define the destination register for the retire phase.

Cost and Delay

The cost and the delay of the circuit **DestCmp** can be estimated as follows (see appendix C.2.2):

$$C(\text{DestComp}) \leq 58 \cdot C_{AND} + 16 \cdot C_{OR} + 27 \cdot C_{MUX},$$

$$D(\text{DestComp}) \leq \max\{2 \cdot D_{OR}, 2 \cdot D_{AND}, D_{MUX}\} + 2 \cdot D_{OR}.$$

4.1.6 Instruction Issue

The circuit **Issue** issues the instructions to the reservation stations. For simplicity a number is assigned to every functional unit type (see table 4.1). The control signal $FU.\star$ computed by the circuit **Decode** are renamed to FU_i , $i \in \{0, \dots, 6\}$, where i is the number of the type of the functional unit (e.g., $FU.IAlu$ is renamed to FU_1). An instruction which uses a functional unit of type i is called “instruction of type i ”.

Due to restrictions in the dispatch order (see section 4.2.2) the processor must have exactly one memory and one branch checking unit. For all other types multiple functional units are possible. The number of functional units of type i for $i \in \{0, \dots, 6\}$ is denoted by n_i . The functional units of type i are denoted by $FU_{i,j}$ for $j \in \{0, \dots, n_i - 1\}$. The total number of functional units n is defined by $n := \sum_{i=0}^6 n_i$.

The processor has one reservation station per functional unit. The reservation station of a functional unit of type i is called “reservation station of type i ”. The reservation station of functional unit $FU_{i,j}$ is denoted by $RS_{i,j}$. All reservation stations of one type have the same number of entries.

An instruction of type i is issued to the first reservation station $RS_{i,j}$ which can accept a new instruction (i.e., $RS_{i,j}.stallOut = 0$). The instruction is sent to that reservation station by asserting its full input. If none of the reservation stations of type i can accept new instructions, issuing has to be stalled.

| no | mnemonic | functional unit |
|----|--------------|------------------------------------|
| 0 | <i>Mem</i> | memory unit |
| 1 | <i>IAlu</i> | integer ALU |
| 2 | <i>IMul</i> | integer multiplicative unit |
| 3 | <i>FAdd</i> | floating point additive unit |
| 4 | <i>FMul</i> | floating point multiplicative unit |
| 5 | <i>FMisc</i> | floating point miscellaneous unit |
| 6 | <i>BC</i> | branch checking unit |

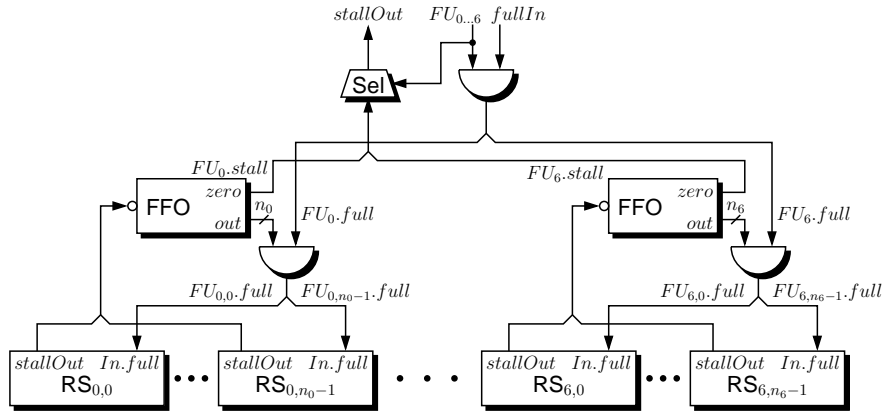
Table 4.1: Functional unit numbers**Figure 4.3:** Computation of stall signal and full bits for issue

Figure 4.3 gives an overview of the circuit **Issue**. It shows the computation of the output stall signal *stallOut* of the circuit **Issue** and the full signals for the functional units $FU_{i,j}.full$. The full signals for the functional units are connected to the full inputs of the corresponding reservation station. The data signals of the instruction to be issued (i.e., *tag*, *operands*, and *control*) are not shown in figure 4.3. They are simply distributed to all reservation stations and connected to the corresponding input busses.

The selection of the reservation station is done as follows: first the signal *fullIn* (indicating that the input registers of the issue circuit contains a valid instruction) is AND-ed with the FU-type indicators FU_i to obtain $FU_i.full$ for $i \in \{0, \dots, 6\}$. The signal $FU_i.full$ indicates that a valid instruction needs to be issued to a reservation station of type i . Next, the number $j \in \{0, \dots, n_i\}$ of the first reservation station of type i that can accept an instruction is determined. This can be done by a find-first-one circuit using the negated stall signals of the reservation stations of type i . The output of the find-first-one circuit is AND-ed to the signal $FU_i.full$ to obtain the full signals for the functional units $FU_{i,j}.full$ for $j \in \{0, \dots, n_i\}$.

The zero output of the find-first-one circuit for the type i indicates that no reservation stations of a type can accept a new instruction. It is used as stall signal for the functional unit type $FU_i.stall$. The overall stall signal for the circuit **Issue** is computed by selecting the stall signal of the type of the instruction to be issued (defined by the FU-type indicator $FU_{0..6}$). This is done by the select circuit in figure 4.3.

If the number of the functional units of a type n_i is one, the find-first-one circuit

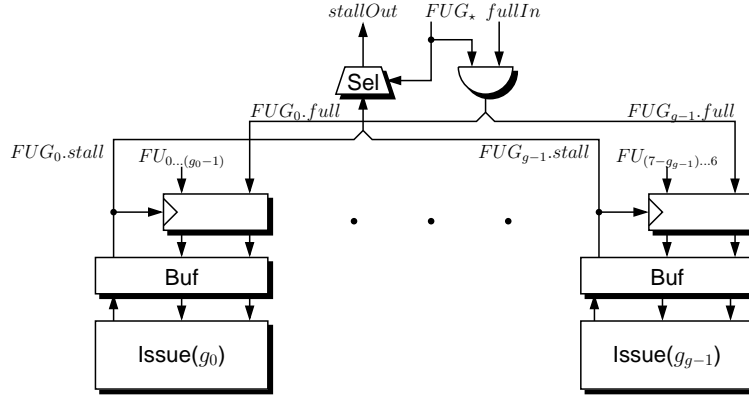


Figure 4.4: Pipelined distribution to the FU types

and the AND gate can be omitted. In this case the full and stall signals for this unit type and reservation station are the same.

Pipelining

The delay of the stall output *stallOut* of the issue circuit is:

$$D(stallOut) \leq \max\left\{\max_{0 \leq i \leq 6} (D(RS_{i,*}.stallOut) + D(FFO(n_i))), D(FU_*)\right\} + D(Sel(7)). \quad (4.1)$$

Assume the processor has two integer ALUs and FP additive units, i.e., $n_1 = n_3 = 2$. The delay of the signal *stallOut* would then be $D(FFO(2)) + D(Sel(7)) = 5$. In order to compute the stall signal for the input register of the issue circuit in one cycle, the delay of the signal *stallOut* may be at most $\delta - D_{AND} = \delta - 1$ (see section 2.5.4). Thus, to support the above values the stage depth must be at least 6.

In order to relax this bound the distribution to the FU types can be split into multiple cycles. For this the FUs types are combined to groups (and sub-groups if the circuit *Issue* is split in more than two cycles). The instructions are first distributed to their group (and sub-group) and then to the functional units. Since the delay of the stall signal is critical, a buffer circuit is placed after the registers for the FU group to decompose the stall computation.

Figure 4.4 shows an overview of a pipelined issue circuit. The FU types are divided into g groups (G_0 to G_{g-1}). The group G_l has g_l members (for l in $\{0, \dots, g-1\}$). For each group G_l a new signal FUG_l has to be computed, indicating whether the instruction has to be issued to a FU type of this group:

$$FUG_l := \bigvee_{i \in G_l} FU_i.$$

These signals can already be computed in the decode circuit. Then, they come directly out of the output registers of the decode sub-phase *D1*. Therefore, it is assumed that the signals FUG_* have the same delay as the signals FU_* .

Let the type i of an instruction be in the group G_k . In the pipelined issue circuit the instructions are first issued to the group G_k . From the group G_k the instruction is sent to the circuit *Issue*(g_k) which needs to support only g_k FU types.

The buffer circuit inserted after the registers for a group decouples the stall signal for the issuing to a group and the stall signals for the issuing from the groups to the reservation stations. Note that the stall signals $FUG_{\star}.stall$ are compute by AND-ing the full bit of the register and the buffer circuit for that group and therefore have delay D_{AND} (see figure 2.4 on page 13). For the sake of readability, this AND-gate is not shown in figure 4.4.

If issuing from the group G_k to the reservation station still cannot be done in one cycle, the groups can be further divided into sub-groups. The circuit $Issue(g_k)$ is then build analogously to the pipelined issue circuit in figure 4.4. If the circuit fits into one cycles it is build analogously to the non-pipelined circuit in figure 4.3.

The computation of the full bit for the groups $FUG_{\star}.full$ and the output stall signal $stallOut$ based on the stall signals of the groups $FUG_{\star}.stall$ happens analogously to the computation for the instruction types in the non-pipelined case. Note that the input signals for the circuits $Issue(\star)$ come out of buffer circuits. For the equation (4.1) then holds $D(FU_{\star}) = D_{MUX}$ (see figure 2.4).

The circuit described above pipelines the issuing of an instruction to its type, but not the issuing from an instruction type to the reservation stations. In order to pipeline this part registers need to be inserted into the computation of the stall signal of the FU types $FU_i.stall$. This would mean that also the computation of the signals $FU_{i,j}.full$ would take several cycles. However, these signals are used inside the reservation station to compute the output stall signals of the reservation stations for the next cycle (see section 4.2.2) and therefore its own value for the next cycle.

Inserting registers into this computation can lead to inconsistencies due to the one cycle dependency, e.g., two instructions could be sent to a reservation station that can only accept one instruction. In order to avoid this problem, issuing from the FU type to the functional units of the type has to be done in one cycles. The bound implicated by this is acceptable.

Cost and Delay

If the size of the group from which an instruction of type i is issued to the reservation stations is one, the full signal for the FU type $FU_i.full$ can be directly derived from the full bit of the group. It does not have to be AND-ed with the FU type indicator FU_i . The delay of the full bit for the group is D_{OR} from the buffer circuit. The find-first-one circuit and the AND-gate at its output are only needed if $n_i > 1$. This reduces the delay of the full signals for the FUs $FU_{i,\star}.full$ to:

$$D(FU_{\star,\star}.full) \leq \max\{D_{OR}, \max_{0 \leq i \leq 6} (D(RS_{i,\star}.stallOut) + D(\text{FFO}(n_i)))\} \\ + \begin{cases} 0 & \text{if } n_i = 1 \\ D_{AND} & \text{if } n_i > 1 \end{cases}.$$

Let e_{RS_i} be the number of entries of the reservation stations of type i . Then the delay of the stall output of these reservation stations is $D(\text{PP-OR}(e_{RS_i}))$ as will later be derived in section 4.2.2. The input full signal to the reservation stations may have at most the delay $\delta - D_{MUX}$ (see section 4.2.1). Since the issuing from the FU type

to the reservation station is not pipelined, n_i and e_{RS_i} are bounded by the following equation:

$$\begin{aligned} \delta &\geq D_{MUX} + D(FU_{i,*}.full) \\ &\geq D_{MUX} + \max\{D_{OR}, \max_{0 \leq i \leq 6} (D(\text{PP-OR}(e_{RS_i})) + D(\text{FFO}(n_i)))\} \\ &\quad + \begin{cases} 0 & \text{if } n_i = 1 \\ D_{AND} & \text{if } n_i > 1 \end{cases} . \end{aligned} \quad (4.2)$$

The input registers of the decode sub-phase $D2$ are not assumed to have buffer circuits. This reduces the delay of the input signal to the issue circuit FU_* to zero. The delay of the output stall signal (see equation (4.1)) may be at most $\delta - D_{AND}$. Let the variable p_I be one, if the circuit **Issue** has to be pipelined. Hence:

$$p_I = \begin{cases} 0 & \text{if } \delta \geq \max_{0 \leq i \leq 6} (D(\text{PP-OR}(e_{RS_i})) + D(\text{FFO}(n_i))) \\ & \quad + D(\text{Sel}(7)) + D_{AND} \\ 1 & \text{else} \end{cases} .$$

If p_I is one, three numbers H , I , and J have to be computed. The number H defines the maximum number of FU types to which an instruction can be issued in one cycle. The maximum number of groups an instruction can be issued to in one cycle is denoted by I . The maximum number of sub-groups an instruction can be issued to from a group is denoted by J .

The delay of the data output of a buffer circuit is D_{MUX} and the delay of the stall output of a buffer circuit is D_{AND} . The delay of the output stall signal of an stage may be at most D_{AND} , as it is computed by an select circuit and therefore the AND-gate of the buffer circuit cannot be merged into the computation (see section 2.5.4). Thus, H , I , and J can be computed using the following equations:

$$\begin{aligned} H &= \max\{h | \delta \geq \max\{\max_{0 \leq i \leq 6} (D(\text{PP-OR}(e_{RS_i})) + D(\text{FFO}(n_i))), D_{MUX}\} \\ &\quad + D(\text{Sel}(h)) + D_{AND}\}, \\ I &= \max\{i | \delta \geq D_{AND} + D(\text{Sel}(i)) + D_{AND}\}, \\ J &= \max\{j | \delta \geq \max\{D_{AND}, D_{MUX}\} + D(\text{Sel}(j)) + D_{AND}\}. \end{aligned}$$

The value of J can be increased using the following optimization. Assume issuing takes c_I cycles. Instead of clearing the issue circuit if the signal *clear* is active, it suffices to clear the reservation stations for c_I cycles when clear is active. After the first clear cycle, the reservation stations do not produce a stall and therefore the instructions in the issue circuit advance in at most $c_I - 1$ cycles to the reservation stations where they are cleared. Note that no valid instruction can enter the reservation stations in the c_I cycles after a clear .

Therefore, it can be assumed, that the registers of the issue circuit do not have to be cleared. Figure 4.5 (a) depicts the computation of the input of the register $FUG.fullBuf$ of a group FUG that issues an instruction to the sub-groups $FUSG_0$ to $FUSG_{J-1}$ without taking the clear signal into account. The figure includes the OR-gate that is used inside the buffer circuit of the group to compute the full output

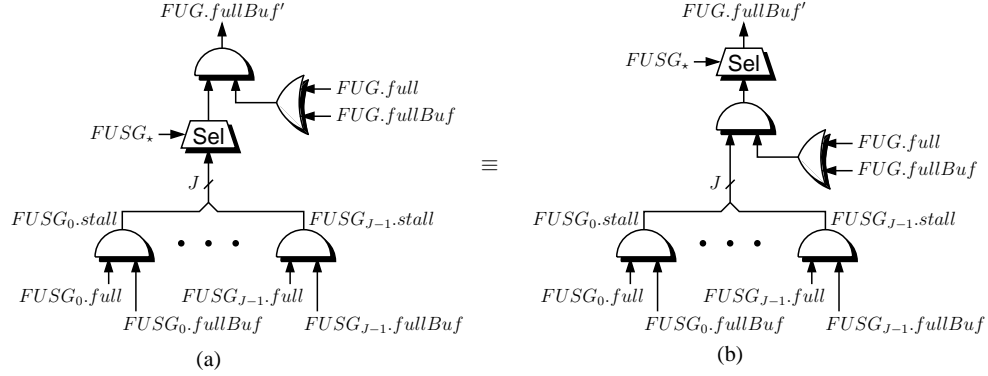


Figure 4.5: Optimized issuing to sub-groups

(see figure 2.4 on page 13. It also includes for each sub-group $FUSG_i$ the AND-gate that is used inside the buffer circuit of the sub-group to compute the stall output $FUSG_i.stall$.

The signals $FUSG_*$ have delay D_{MUX} since they are part of the data output of the buffer circuit for the group FUG (see figure 2.4). All other input signals in the figure come directly out of registers. Hence, the delay of the signal $FUG.fullBuf'$ can be reduced by moving the last AND-gate of the computation below the select circuit (see figure 4.5 (b)). Then the value of J can be computed as:

$$J = \max\{j | \delta \geq \max\{2 \cdot D_{AND}, D_{AND} + D_{OR}, D_{MUX}\} + D(\text{Sel}(j))\}.$$

In order to reduce the number of FU types per groups I and J must be at least 2. Thus, it must hold:

$$\delta \geq D_{AND} + D(\text{Sel}(2)) + D_{AND}, \quad (4.3)$$

$$\delta \geq \max\{2 \cdot D_{AND}, D_{AND} + D_{MUX}, D_{MUX}\} + D(\text{Sel}(2)). \quad (4.4)$$

Both equations hold for $\delta \geq 5$.

The number of cycles needed for the circuit **Issue** c_I can be computed as follows:

$$c_I = \begin{cases} 1 & \text{if } p_I = 0 \\ 2 & \text{if } p_I = 1 \wedge H \cdot I \geq 7 \\ 3 & \text{if } p_I = 1 \wedge H \cdot I < 7 \wedge H \cdot I \cdot J \geq 7 \\ 4 & \text{if } p_I = 1 \wedge H \cdot I \cdot J < 7 \end{cases}.$$

The delay of the output stall signal is:

$$D(stallOut) \leq \begin{cases} \max_{0 \leq i \leq 6} (D(\text{PP-OR}(e_{RS_i})) + D(\text{FFO}(n_i))) + D(\text{Sel}(7)) & \text{if } c_I = 1 \\ D_{AND} + D(\text{Sel}(\lceil 7/G \rceil)) & \text{if } c_I = 2 \\ D_{AND} + D(\text{Sel}(\lceil 7/(G \cdot H) \rceil)) & \text{if } c_I = 3 \\ D_{AND} + D(\text{Sel}(\lceil 7/(G \cdot H \cdot I) \rceil)) & \text{if } c_I = 4 \end{cases}.$$

The inputs to the circuit **Issue** are the full bit, the tag, the operands, and the control bits. The width of the control bits for the functional units except the memory and the

branch checking units is approximated by 8. The memory unit needs an additional 16 bit immediate constant. The branch checking unit needs and 24 bit immediate constant, the PC of the instruction, the predicted branch target, and the way of the branch target buffer as additional control signals (see chapter 6). Thus, the number of input I_I and outputs O_I of the issue circuit are:

$$\begin{aligned} I_I &= 1 + 7 \cdot l_{ROB} + 64 + 64 + 5 + 2 + 8 + 24 + 32 + 32 + k_{BTB}, \\ O_I &= n \cdot (1 + 3 \cdot l_{ROB} + 64 + 8) + (n_3 + n_4 + n_5) \cdot (4 \cdot l_{ROB} + 64 + 5) \\ &\quad + 16 + 24 + 32 + 32 + k_{BTB}. \end{aligned}$$

Then the approximated cost of the issue circuit is:

$$\begin{aligned} C(\text{Issue}) &\leq \sum_{i=0}^6 (C(\text{FFO}(n_i)) + n_i \cdot C_{AND}) + 7 \cdot C_{AND} + C(\text{Sel}(7)) \\ &\quad + (c_I - 1) \cdot \lceil (I_I + O_I)/2 \rceil \cdot C_{REG}. \end{aligned}$$

4.1.7 Stalling

The first stage of the sub-phase $D1$ has to be stalled if the ROB is full or if decoding has to be stopped until a detected branch misprediction has retired. The ROB is full if the signal $ROB.full$ computed by the ROB control (see section 4.6.5) is active. Decoding has to be stopped due to a misprediction if the signal $IF.haltdec$ computed by the instruction register environment (see section 6.4) is active. Thus:

$$D1.genStall^0 := ROB.full \vee IR.haltdec, \quad (4.5)$$

$$D1.stall^0 := D1.full^0 \wedge (D1.stallIn^0 \vee D1.genStall^0). \quad (4.6)$$

An instruction which reads the special register $IEEEf$ (indicated by the control signal $readIEEEf$ computed in the circuit **Decode**) must wait until all floating point operations have retired. This is due to the fact that all floating point instructions write the register $IEEEf$ implicitly. For simplicity the instructions reading that register wait until all preceding instructions have retired, which is indicated by the signal $allRet$ computed by the ROB control. Instructions reading the register $IEEEf$ are considered rare, hence the performance impact can be neglected.

The instructions which read the register $IEEEf$ must wait in the pipeline stage of the decode sub-phase $D1$ in which the register files in the circuit **RF** return the result of the read access (denoted by c_{RF}). The SPR guarantees that the correct content of register $IEEEf$ is returned as soon as the signal $allRet$ is active without restarting the read access (see section 4.7.4). Thus:

$$D1.genStall^{c_{RF}} := readIEEEf \wedge \overline{allRet}, \quad (4.7)$$

$$D1.stall^{c_{RF}} := D1.full^{c_{RF}} \wedge (D1.stallIn^{c_{RF}} \vee D1.genStall^{c_{RF}}). \quad (4.8)$$

As discussed in the overview section, the ROB access in sub-phase $D2$ must not finish before the corresponding instruction has been issued. Otherwise the instruction would miss the result of the ROB access. If issuing is done in one cycle, this can be

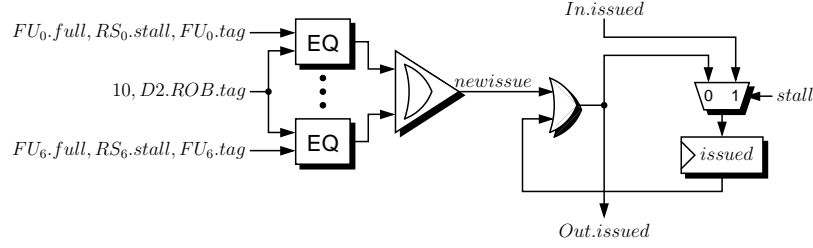


Figure 4.6: Issue test

guaranteed without extra hardware. It suffices that the output stall signal of the issue circuit is used as stall signal of the input register of the sub-phase $D2$:

$$D2.stall^0 := D2.full^0 \wedge Issue.stallOut. \quad (4.9)$$

An instruction then waits in the input register of the decode sub-phase $D1$ until it is issued. The ROB access does not have to be stalled.

If issuing is pipelined additional hardware is needed to detect whether an instruction has already been issued. For each stage of the ROB access, a circuit **IssueTest** (see figure 4.6) checks if the instruction which is in this stage is already issued to a reservation station.

An instruction is issued to a functional unit of type i if the stall signal for this type $RS_i.stall$ is not active and the full bit for this type $FU_i.full$ is active (see section 4.1.6). To check whether the issued instruction is the instruction in the ROB access stage, the tags are compared. The tag of the instruction in the ROB access stage is called $D2.ROB.tag$ in the figure. The signal $newissue$ is active if the instruction in the ROB access stage is issued to any functional unit. The values of $newissue$ are accumulated in the register $issued$, i.e., $issued$ is active if the instruction has been issued in any previous cycle.

The register $issued$ has to be updated even if the stage of the ROB access is stalled. The update of this register is done analogously to the update of the register $forw$ in the forwarding circuit with stalling (see section 2.6.3), i.e., if the ROB access stage is stalled, the register $issued$ is updated, otherwise the updated result is saved in the next stage. The computation of $newissue$ can be pipelined similar to the computation of $forw$ in the forward circuit with stalling.

Let c_{D2} be the number of stages of the ROB read access in the decode sub-phase $D2$. The output register of the ROB access has to be stalled if it is full and the register $issued$ is not active:

$$D2.stall^{c_{D2}} := D2.full^{c_{D2}} \wedge \overline{D2.issued^{c_{D2}}}.$$

This guarantees that the ROB read access of an instruction is held in the last stage of the access until the instruction is in the reservation station and snoops on the output of the ROB. The other stages are stalled as usual, i.e., $stallOut := stallIn \wedge full$. Thus, the stall output of the ROB access $ROB.stallOut$ can be computed as:

$$ROB.stallOut := \bigwedge_{i=1}^{c_{D2}} D2.full^i \wedge \overline{D2.issued^{c_{D2}}} \quad (4.10)$$

The input register of the decode sub-phase $D2$ is stalled if either the ROB access or the issue circuit produce a stall:

$$D2.stall^0 := D2.full^0 \wedge (ROB.stallOut \vee Issue.stallOut). \quad (4.11)$$

Let c_I be the number of cycles needed for issuing. Then, combining formulas 4.9 and 4.11 one gets for stall output of the decode sub-phase $D2$ (which is the stall signal of the first stage of $D2$):

$$D2.stallOut := \begin{cases} D2.full^0 \wedge ROB.stallOut & \text{if } c_I = 1 \\ D2.full^0 \wedge (ROB.stallOut \vee Issue.stallOut) & \text{if } c_I > 1 \end{cases} \quad (4.12)$$

Let l_{ROB} be the width of the tags. The the delay of the circuit **IssueTest** is:

$$D(\text{IssueTest}) \leq D(\text{EQ}(l_{ROB} + 1)) + D(\text{OR-Tree}(7)) + D_{OR}.$$

Let c_{IT} be the number of cycles needed for the circuit **IssueTest**. The number of inputs and outputs are $8 \cdot l_{ROB} + 15$ respectively 1. It holds

$$\begin{aligned} c_{IT} &= \lceil D(\text{IssueTest}) / (\delta - D_{MUX}) \rceil, \\ C(\text{Issue Test}) &\leq 7 \cdot C(\text{EQ}(l_{ROB} + 1)) + 8 \cdot C_{OR} + C_{MUX} + C_{REG} \\ &\quad + c_{IT} \cdot (4 \cdot l_{ROB} + 8) \cdot (C_{REG} + C_{MUX}). \end{aligned}$$

4.1.8 Cost and Delay

The delay of the decode phase depends on the number of buffer circuits that have to be inserted in order to compute the stall signals within the cycle time. Up to two buffer circuits are inserted in order to reduce the delay of the stall signals. The first buffer circuit is inserted after the stage c_{RF} in which the register files return the results. The second buffer circuit is inserted somewhere before the stage c_{RF} . To further decrease the delay of the stall signal, more buffer circuits could be inserted into the stages of $D1$ and the ROB access in $D2$; however, this is not necessary for a stage depth of 5 and greater.

If no buffer circuit has to be inserted into the decode sub-phase $D1$, its delay is:

$$D(D1) \leq \max\{D(\text{PT}), D(\text{RF}), D(\text{Decode})\} + D(\text{OpGen}).$$

The values for the number of cycles of the sub-phase $D1$ c_{D1} and for c_{RF} can be computed based on the number of buffer circuits b :

$$\begin{aligned} c_{D1}(b) &= \lceil (D(D1) + b \cdot D_{MUX}) / \delta \rceil, \\ c_{RF}(b) &= \begin{cases} \lceil D(\text{SPR-RF}) / \delta \rceil & \text{if } b \leq 1 \\ \lceil (D(\text{SPR-RF}) + D_{MUX}) / \delta \rceil & \text{if } b > 1 \end{cases}. \end{aligned}$$

If no buffer circuits are inserted in the sub-phase $D1$, the stall signal for the first stage of $D1$ (and therefore the output stall signal of the whole decode phase $D1.stallOut$) is computed as (see equation 4.6):

$$\begin{aligned} D1.stallOut &= D1.full^0 \wedge (D1.stallIn^0 \vee D1.genStall^0) \\ &\stackrel{(4.5)}{=} D1.full^0 \wedge (D1.stallIn^0 \vee ROB.full \vee IR.haltdec). \end{aligned}$$

The stages 1 to $c_{RF} - 1$ do not generate stall signals. Thus:

$$D1.stallOut = D1.full^0 \wedge ((\bigwedge_{i=1}^{c_{RF}-1} D1.full^i \wedge D1.stallIn^{c_{RF}-1}) \vee ROB.full \vee IR.haltdec).$$

This can be transformed using the distributive law to:

$$D1.stallOut = (\bigwedge_{i=0}^{c_{RF}-1} D1.full^i \wedge D1.stallIn^{c_{RF}-1}) \vee (D1.full^0 \wedge (ROB.full \vee IR.haltdec)).$$

Using formula 4.8 for the computation of $D1.stall^{c_{RF}} = D1.stallIn^{c_{RF}-1}$ it follows:

$$\begin{aligned} D1.stallOut &= (\bigwedge_{i=0}^{c_{RF}} D1.full^i \wedge (D1.stallIn^{c_{RF}} \vee D1.genStall^{c_{RF}})) \\ &\vee (D1.full^0 \wedge (ROB.full \vee IR.haltdec)) \\ &\stackrel{(4.7)}{=} (\bigwedge_{i=0}^{c_{RF}} D1.full^i \wedge (D1.stallIn^{c_{RF}} \vee (readIEEEf \wedge \overline{allRet}))) \\ &\vee (D1.full^0 \wedge (ROB.full \vee IR.haltdec)) \end{aligned} \quad (4.13)$$

The remaining stages of the sub-phase $D1$ do not generate stalls. Let c_{D1} be the number of stages of $D1$. Then, if no buffer circuits are inserted in $D1$, the stall output of the decode phase can be computed as:

$$\begin{aligned} D1.stallOut &= (\bigwedge_{i=0}^{c_{RF}} D1.full^i \wedge ((\bigwedge_{j=c_{RF}+1}^{c_{D1}-1} D1.full^j \wedge D2.stallOut) \\ &\vee (readIEEEf \wedge \overline{allRet}))) \\ &\vee (D1.full^0 \wedge (ROB.full \vee IR.haltdec)) \end{aligned}$$

Figure 4.7 shows the computation of the signal $D1.stallOut$.

Let c_I be the number of stages needed for the circuit **Issue** and c_{D2} the number of cycles needed for the ROB access in the decode sub-phase $D2$. The value of c_{D2} is computed in the ROB section 4.6. Then the delay of the output stall signal for the decode sub-phase $D2$ can be computed as:

$$\begin{aligned} D(ROB.stallOut) &\stackrel{(4.10)}{\leq} D(\text{AND-Tree}(c_{D2})), \\ D(D2.stallOut) &\stackrel{(4.12)}{\leq} \begin{cases} D(Issue.stallOut) & \text{if } c_I = 1 \\ \max\{D(ROB.stallOut), \\ D(Issue.stallOut)\} + D_{OR} & \text{if } c_I > 1 \end{cases} \end{aligned}$$

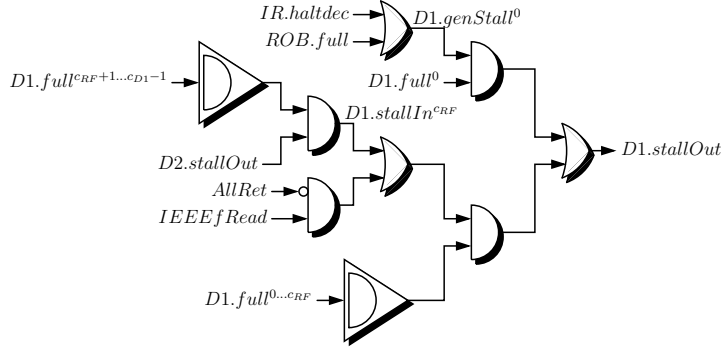


Figure 4.7: Computation of the decode stall output without buffer circuit

The control signals *AllRet*, *IEEEfRead*, and *IR.haltdec* come directly out of registers and therefore have delay zero. The signal *ROB.full* has delay D_{MUX} as discussed later in the section 4.6.7. The stall output of the decode phase may at most have the delay $\delta - D_{MUX}$ (see section 6.4). Thus, the number of buffer circuits b can be zero if the following equation holds:

$$\begin{aligned} \delta - D_{MUX} \geq & \max\{\max\{D(D2.stallOut), D(\text{AND-Tree}(c_{D1}(0) - c_{RF}(0) - 1))\} \\ & + D_{AND} + D_{OR}, \\ & D(\text{AND-Tree}(c_{RF}(0) + 1)), D(ROB.full) + D_{OR}\} \\ & + D_{AND} + D_{OR}. \end{aligned}$$

The first buffer circuit is inserted in the stage after the output of the register files return the result of the read accesses, i.e., $c_{RF}(1) + 1$. This allows to split the stall computation at the signal $D1.stall^{c_{RF}+1} = D1.stallIn^{c_{RF}}$. In equation 4.13 the signal $D1.stallIn^{c_{RF}}$ can then be computed as (see figure 2.4 on page 13):

$$D1.stallIn^{c_{RF}} = D1.full^{c_{RF}+1} \vee D1.fullBuf^{c_{RF}+1}.$$

The critical signal of the stall computation of the sub-phase *D1* for the stages below the stage c_{RF} is then the input of the full bit of the buffer circuit $D1.fullBuf^{c_{RF}+1}$. The AND-gate in the buffer circuit that is needed for the computation of this signal (see figure 2.4) can be incorporated into the AND-Tree for the signals $D1.full^{c_{RF}+1...c_{D1}-1}$. This increases the number of inputs by 2 (see section 2.5.4). Figure 4.8 shows the computation of the critical signals of the stall computation if one buffer circuit is inserted.

Thus, one buffer circuit is sufficient ($b = 1$) if the following equations holds:

$$\begin{aligned} \delta \geq & \max\{D(D2.stallOut), D(\text{AND-Tree}(c_{D1}(1) - c_{RF}(1) + 1)) + D_{AND}\}, \\ \delta \geq & \max\{D_{AND} + D_{OR}, D(\text{AND-Tree}(c_{RF}(1) + 1)), D_{MUX} + D_{OR}\} \\ & + D_{AND} + D_{OR} + D_{MUX}. \end{aligned}$$

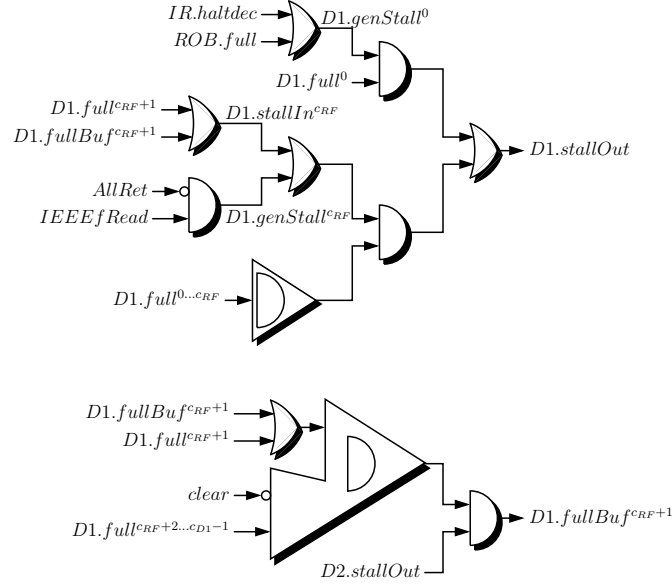


Figure 4.8: Computation of the decode stall signal with one buffer circuit inserted

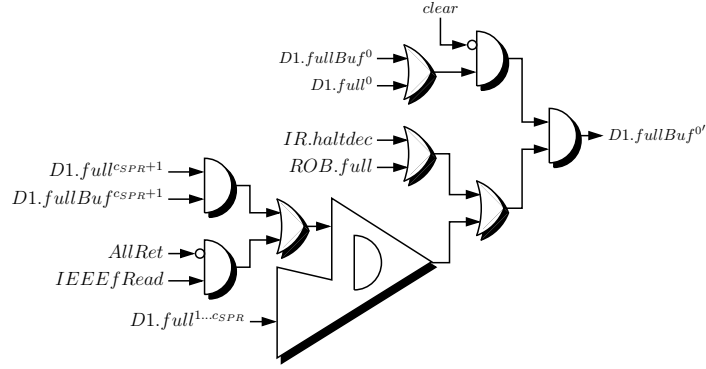


Figure 4.9: Computation of the signal $D1.fullBuf^{0'}$ with two buffer circuits inserted

The AND-gate and the two OR-gates needed for the computation of OR of the signals $D1.stallIn^{c_{RF}}$ and $D1.genStall^{c_{RF}}$ (see figure 4.8) can be merged into the AND-tree below. This effectively increases the number of inputs of the AND-tree by 4 and allows the balancing of the AND-tree in order to minimize the overall delay. Thus, the second equation can be replaced by:

$$\delta \geq \max\{D(\text{AND-Tree}(c_{RF}(1) + 5)), D_{MUX} + D_{OR} + D_{AND}\} \\ + D_{OR} + D_{MUX}.$$

If δ does not fulfill the last equation a second buffer circuit is inserted into stage 0 of $D1$. The the delay of the stall output of the decode phase is D_{AND} since it comes directly out of a buffer circuit. Thus, for $\delta \geq 5$, the requirement $D(D1.stallOut) \leq \delta - D_{MUX}$ (see section 6.4) holds.

Figure 4.9 depicts the computation of the input $D1.fullBuf^{0'}$ of the full register of the buffer circuit in stage 0 of sub-phase $D1$ if two buffer circuits are inserted into $D1$. Using the trick presented in figure 2.5 on page 17 the last AND-gate of the circuit

can be removed from the critical path. This adds 3 more inputs to the AND-tree. Thus, if two buffer circuits are inserted, the following equations must hold:

$$\begin{aligned}\delta &\geq \max\{D(D2.stallOut), D(\text{AND-Tree}(c_{D1}(2) - c_{RF}(2) + 1)) + D_{AND}\}, \\ \delta &\geq \max\{D(\text{AND-Tree}(c_{RF}(2) + 7)), D_{MUX} + D_{OR} + D_{AND}\} + D_{OR}.\end{aligned}$$

These equations hold for $\delta = 5$.

The inputs of the decode sub-phase $D1$ consist of the full bit, the instruction register, and the signals used by the branch prediction. The width of the input is $100 + k_{BTB}$ (the full bit, the 32 bit instruction word, the 32 bit instruction address, the 32 bit prediction result, two interrupt signals, a control signal, and k_{BTB} bits for the branch target buffer way, see section 6.9). The outputs of the decode sub-phase $D1$ consist of the inputs to the issue circuit and the data written into the ROB in sub-phase $D2$. The width of the output is $I_I + 15$. The total cost for the decode sub-phase $D1$ is (excluding the RAM environments):

$$\begin{aligned}C(D1) &\leq C(\text{Decode}) + C(\text{DestCmp}) + C(\text{OPGen}) \\ &\quad + c_{D1} \cdot \lceil (115 + k_{BTB} + I_I)/2 \rceil \cdot C_{REG}.\end{aligned}$$

Let c_I be the number stages of the circuit **Issue** and let c_{D2} be the number of stages of the read access to the ROB during $D2$. The cost of the sub-phase $D2$ not counting the ROB environment is:

$$\begin{aligned}C(D2) &\leq C(\text{Issue}) + (I_I + 15) \cdot C_{REG} \\ &\quad + \begin{cases} 0 & \text{if } c_I = 1 \\ c_{D2} \cdot C(\text{IssueTest}) & \text{if } c_I > 1 \end{cases}.\end{aligned}$$

4.2 Dispatch

As described in chapter 3 each functional unit has one reservation station of one or more entries. The instructions wait in the reservation stations until all operands are valid. The operands get valid by snooping on the CDB and on the output of the ROB access started during decode. The snooping on the ROB output is necessary since the instructions are issued before the ROB access. As soon as all operands are valid, the instruction is sent to the functional unit and the entry holding the instruction is cleared.

Figure 4.10 shows a reservation station with e_{RS} entries. Each entry can hold one instruction. The circuits for the entries **RS-Entry** _{k} ($k \in \{0, \dots, e_{RS} - 1\}$) form a queue. New instructions are always filled into entry 0. Whenever possible instructions move to the next entry to make room for new instructions in entry 0. Since instructions cannot overtake each other in the reservation station, the oldest instruction is always in the entry with the highest number.

The busses $CDB.\star$ and $D2.OP\star.ROB.\star$ are connected to every entry. If the tag on one of the busses equals the tag of a not already valid operand of an entry, the operand in the entry is updated. The data on the bus is saved and the operand is marked valid. An arbitration of the two busses is not necessary, as they cannot update the same

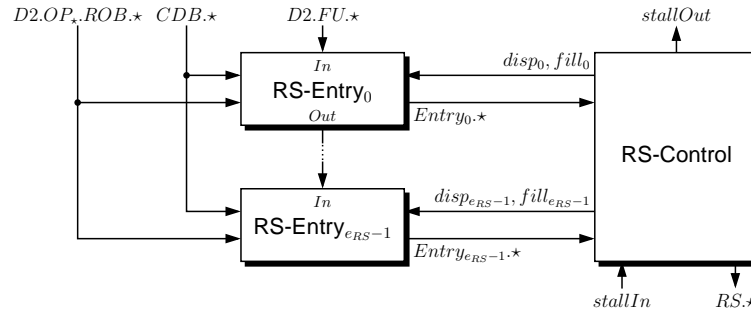


Figure 4.10: Reservation station

operand in the same cycle, since otherwise the instruction that is forwarded would be on the CDB and in the ROB at the same time.

The reservation station is controlled by the circuit **RS-Control**. This circuit selects the oldest instruction which is ready (i.e., all operands are valid) and sends it to the functional unit via the bus RS_* . This step is called dispatch of an instruction. The reservation station control also controls the movement of the entries in the queue and computes the output stall signal of the reservation station.

4.2.1 Entries

The circuit **RS-Entry** for one reservation station entry is depicted in figure 4.11. The input bus In_* equals the output bus of the preceding entry respectively the input received from the issue circuit in case of the entry 0. The output bus Out_* contains the updated content of the entry. If the control signal $fill$ is active, the entry is filled with the content of the bus In_* . Otherwise the current content is updated and held in that stage.

The register $full$ indicates that the entry contains a valid instruction. It is set by filling a valid instruction into the entry via the bus In_* . The $full$ signal is reset if the instruction is dispatched to the functional unit (indicated by $disp = 1$) or if the reservation station is cleared ($clear = 1$). The register con contains information about the instruction which are not altered by the reservation station, for example the tag and the opcode of the instruction. Hence, it is only updated if $fill = 1$.

The circuit **RS-Entry** has a sub-circuit **RS-Op** for every operand. Each operand has a valid bit, a tag and a data field. The number of operands o depends of the type of the functional unit. The maximum number of operands is six for the floating point units. Table 4.2 maps the operands to the **RS-Op** circuits of the reservation station entries for the functional units.

The operands are updated by the CDB bus CDB_* and the output of the ROB read access which was started in decode sub-phase $D2$. Note that the ROB is accessed for every operand in parallel and therefore computes one separate bus for every operand (see section 4.6). Similar to the forwarding circuit with stalling the CDB and the ROB update the output bus Out_* of the entry. Thus, if the instruction flows from entry i to $i + 1$ the entry $i + 1$ is updated. If the instruction remains in entry i , this entry is updated.

The bus $Entry_*$ is sent to the control circuit **RS-Control**. The content of this bus

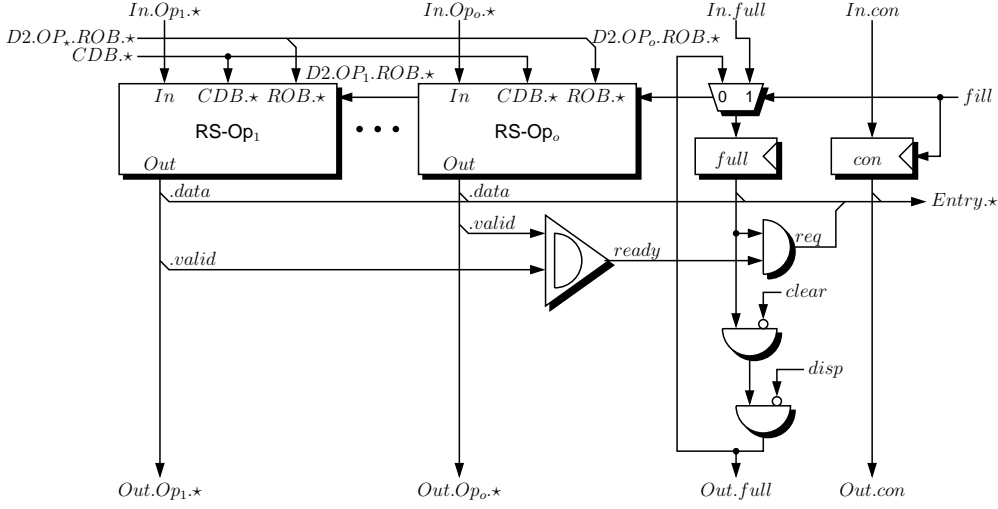


Figure 4.11: Reservation station entry

| | RS-Op ₁ | RS-Op ₂ | RS-Op ₃ | RS-Op ₄ | RS-Op ₅ | RS-Op ₆ |
|---------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| width | 32 | 32 | 32 | 32 | 5 | 2 |
| Mem | $OP_1.lo$ | $OP_2.lo$ | | | | |
| IAlu | $OP_1.lo$ | $OP_2.lo$ | | | | |
| IMulDiv | $OP_1.lo$ | $OP_2.lo$ | | | | |
| FAdd | $OP_1.hi$ | $OP_1.lo$ | $OP_2.hi$ | $OP_2.lo$ | OP_3 | OP_4 |
| FMulDiv | $OP_1.hi$ | $OP_1.lo$ | $OP_2.hi$ | $OP_2.lo$ | OP_3 | OP_4 |
| FMisc | $OP_1.hi$ | $OP_1.lo$ | $OP_2.hi$ | $OP_2.lo$ | OP_3 | OP_4 |
| BCU | $OP_1.lo$ | $OP_2.lo$ | | | | |

Table 4.2: Mapping of the operands

is the same as the content of the bus $Out.★$ with two differences: first, the full bit of the bus $Entry$ holds the content of the register $full$ instead of the updated value. This old value of the full bit is used to decide whether new informations can be filled into the entry (see section 4.2.2). Second, the bus $Entry.★$ has an extra bit req indicating that the entry contains a valid and ready (i.e., all operands are valid) instruction and hence requests that the instruction in this entry gets dispatched:

$$ready := \bigwedge_{j=1}^o Op_j.valid,$$

$$req := full \wedge ready.$$

Let w_C be the width of the register con . Then the cost of an entry with o operands with width w_1 to w_o can be estimated as:

$$C(\text{RS-Entry}(w_C, o, w_★)) \leq \sum_{j=1}^o C(\text{RS-Op}(w_j)) + C(\text{AND-Tree}(o))$$

$$+ 2 \cdot C_{AND} + C_{MUX} + (w_C + 1) \cdot C_{REG}.$$

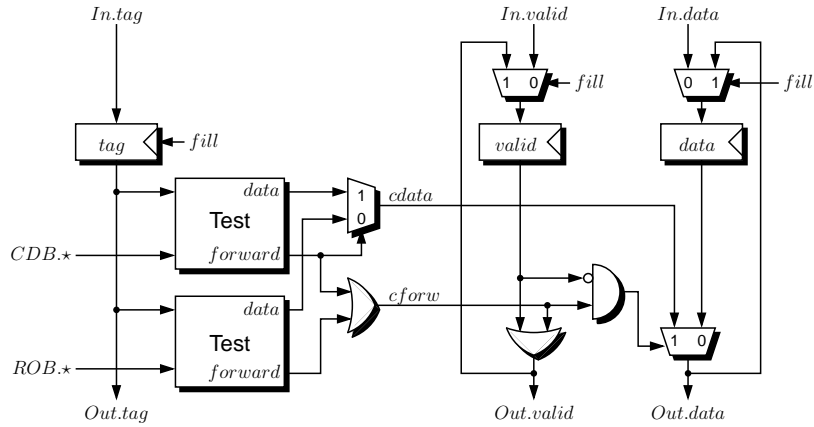


Figure 4.12: Single operand of a reservation station entry

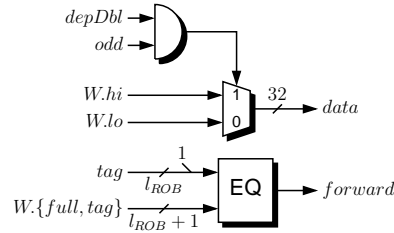


Figure 4.13: Modified test circuit for low part operands reading double precision results

Operands

Figure 4.12 shows the operand circuit RS-Op. The circuit saves the valid bit, the tag and the data of the operand in the respective registers. If the *fill* signal is active, the content of the input bus *In.** is saved into the registers, otherwise the current operand is held in the entry, i.e., the output bus *Out.** with the updated values for the operand is latched.

Similar to the RAM forwarding circuit in section 2.6.1 the sub-circuits **Test** compares the tag of the busses *CDB.** and *ROB.** with the tag of the operand. If the tags match, the signals *CDB.full* respectively *ROB.full* indicate valid forwarding data, and the operand is not yet valid, the respective forwarding data are multiplexed into the data output *Out.data*. The new valid signal is computed as OR of the old valid and the forward signals from the two test circuits.

Assume a 32 bit instruction uses the high part of a 64 bit result as operand. This can happen if the 32 bit instruction depends on a 64 bit result and reads an odd register. 64 bit results are either written into the floating point register file or the registers 8 and 9 of the special purpose register file (integer multiplication / division instructions) file as target. The high part of these registers can only be used by the operand 2 of the memory unit and the integer ALU, and operands 2 and 4 of the floating point units.

For these operands, the circuit **Test** is replaced by a modified circuit that uses the high part of the CDB respectively the ROB output if the instruction reads an odd address and the depends on 64 bit result (see figure 4.13). The signal *depDbl* indicates that the instruction that the operand depends on is a double precision instruction, the

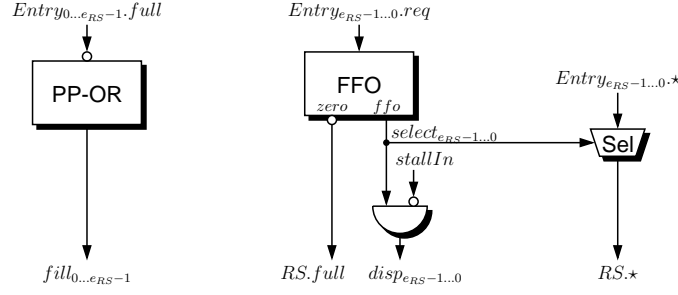


Figure 4.14: Reservation station control

signal *odd* is active if the operand address is odd. These two signals are computed during decode (see section 4.1.4) and are part of the control bus *con* of the reservation station entries.

The cost of an operand circuit with w data bits can be estimated as:

$$C(\text{RS-Op}(w)) \leq 2 \cdot C(\text{Test}(l_{ROB})) + 3 \cdot w \cdot C_{MUX} + w \cdot C_{REG} \\ + 2 \cdot C_{OR} + C_{AND} + C_{MUX} + (1 + l_{ROB}) \cdot C_{REG}.$$

The additional costs for the entries of a reservation station of type i ($i = 2$ for an integer ALU reservation station, $i \in \{4, 5, 6\}$ for floating point reservation stations) are:

$$C(\text{RS-Entry}(o, w_*, w_C))^+ \leq \begin{cases} 32 \cdot C_{MUX} + C_{AND} & \text{if } i = 1 \\ 2 \cdot (32 \cdot C_{MUX} + C_{AND}) & \text{if } i \in \{3, 4, 5\} \\ 0 & \text{else} \end{cases}$$

4.2.2 Reservation Station Control

The reservation station control circuit RS-Control (see figure 4.14) computes the control signals $disp_*$ and $fill_*$ for the entries, the output $RS.★$ of the reservation station to the functional unit, and the stall signal *stallOut* to the issue circuit.

The entry k can be filled if the entry is not full ($full_k$) or the content is filled into the next entry ($fill_{k+1}$). The reservation station can accept new data if the first entry may be filled. Since the last entry (number $e_{RS} - 1$) cannot be filled into any other entry, it follows:

$$fill_k := \bigvee_{j=k}^{e_{RS}-1} \overline{full_j}, \\ stallOut := \overline{fill_0}.$$

Note that the stall output only depends on the full bits of the entries. Hence, the reservation station splits the stall signal similar to the buffer circuit. The drawback is that the issue circuit may be stalled even if an instruction is currently dispatched to the functional unit and therefore its entry could be filled. Taking the dispatch signals into account for the computation of the fill and the stall signals would significantly increase the delay of the stall output. This could make it necessary to add a buffer

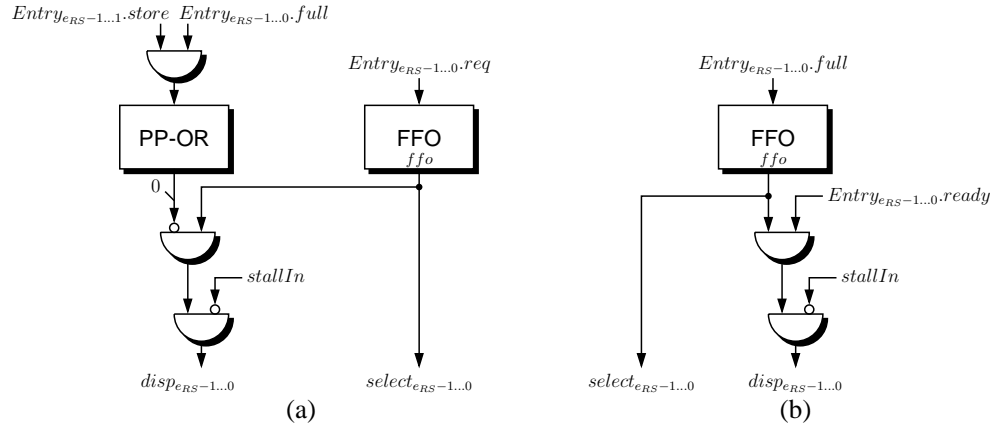


Figure 4.15: Dispatch computation for memory (a) and branch checking (b) unit

circuit above the reservation station. Instead one could just increase the size of the reservation station by one entry to get the same effect.

If the stall input *StallIn* from the functional unit is not active, the control circuit dispatches the oldest instruction in the reservation station which is ready to the functional unit. The entry k contains a valid and ready instruction if the signal request signal of this entry $Entry_k.req$ is active. A find-first-one circuit FFO using the request signals computes as output ffo the entry k with the highest index that contains a valid and ready instruction. Hence, the signals $select_*$ computed from the output ffo unary select the entry containing the oldest ready instruction. The negation of the stall input $stallIn$ is AND-ed to this output obtaining the dispatch signals $disp_*$. Thus, no instruction is dispatched if the stall input is active.

The output *zero* of the circuit FFO indicates that no entry is full and ready. If this signal is not active, a valid instruction can be sent to the functional unit. Thus, negation of the *zero* output can be used as *full* signal for the output bus $RS.*$. The signals $select_*$ are used to select from the entry outputs $Entry_{*,*}$ the instruction which is sent to the functional unit via the output bus $RS.*$.

For the memory unit and the branch checking unit small changes have to be made to the computation of the dispatch signals due to restrictions of the dispatch order. Note that due to these restrictions the $DLX_{\pi+}$ must have exactly one memory and branch checking unit.

The reservation station of the memory unit has to guarantee that no memory instruction overtakes a store instruction, as the store may write to the same address. Note that the address of a memory access is computed inside the memory unit and therefore not known by the memory reservation station. A memory instruction may only be dispatched if no older instruction is a store instruction. In order to check for all entries whether an older instruction is a store instruction, a parallel prefix OR of the signals $Entry_{*,store} \wedge Entry_{*,full}$ indicating a valid store instruction is computed. The output of the parallel prefix OR is used to turn off the dispatch signal as computed from the find first one circuit (see figure 4.15 (a)).

The branch checking unit can check branches only in-order. Hence, the reservation station of the branch checking unit must dispatch the instructions in order. Only the

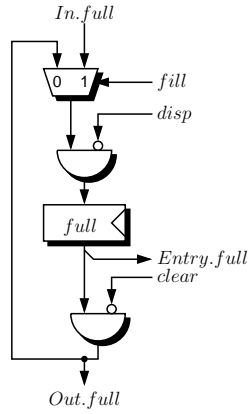


Figure 4.16: Full bit computation for one entry

oldest valid instruction is checked if it is ready. The oldest valid instruction can be computed using a find-first-one circuit on the full bits. The output of the find-first-one circuit is AND-ed with the ready bits in order to check whether the oldest instruction is actually ready (see figure 4.15 (b)). Since only the oldest instruction may be dispatched to the branch checking unit the signals $select_*$ can be derived directly from the output of the find-first-one circuit.

If the number of entries e_{RS} is 1, the requirements for memory and branch checking unit are automatically fulfilled. Thus, no special circuit is needed if e_{RS} is 1. Additionally the signals $fill$ and $disp$ cannot be active at the same time (the full bit must be inactive for $fill$ and active for $disp$). This allows to move the AND gate which resets the full bit on dispatch in figure 4.11 below the multiplexer controlled by the fill signal (see figure 4.16). Hence, the delay of the path through the dispatch signal is further reduced.

The cost of the control circuit RS-Control for a reservation station of type i with o operands of width w_1 to w_o and w_C control bits is:

$$\begin{aligned}
 C(\text{RS-Control}(e_{RS}, o, w_*, w_C)) &\leq C(\text{PP-OR}(e_{RS})) + C(\text{FFO}(e_{RS})) \\
 &\quad + 2 \cdot e_{RS} \cdot C_{AND} \\
 &\quad + \left(\sum_{j=0}^o w_j + w_C + 1 \right) \cdot C(\text{Sel}(e_{RS})) \\
 &\quad + \begin{cases} C(\text{PP-OR}(e_{RS} - 1)) & \text{if } i = 0 \\ +e_{RS} \cdot C_{AND} & \\ 0 & \text{if } i \neq 0 \end{cases} .
 \end{aligned}$$

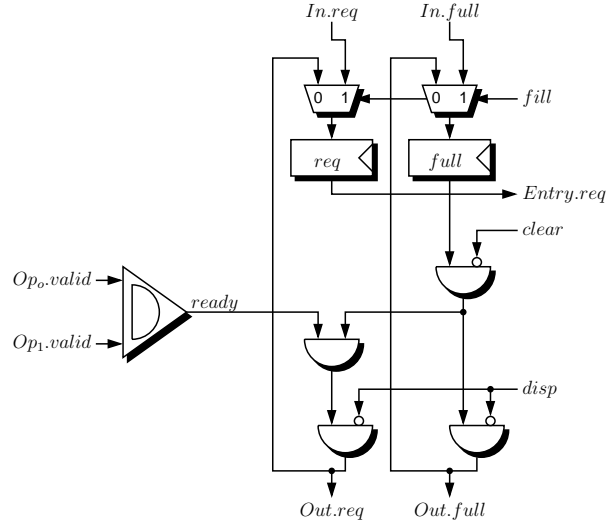


Figure 4.17: Inserting a register after the request bit

4.2.3 Pipelining

For a integer ALU reservation station of type with e_{RS} entries, the delay of the path from the busses $CDB.*$ and $ROB.*$ to the full bit is:

$$\begin{aligned}
 D(\{ROB, CDB\}.* \rightsquigarrow Entry_{*}.full') &\leq D(\text{Test}(l_{ROB})) + 2 \cdot D_{OR} \\
 &\quad (Out.valid, \text{figure 4.12}) \\
 &\quad + D(\text{AND-TREE}(2)) \\
 &\quad \quad (ready, \text{figure 4.11}) \\
 &\quad + D_{AND} \quad (req, \text{figure 4.11}) \\
 &\quad + D(\text{FFO}(e_{RS})) + D_{AND} \\
 &\quad \quad (disp_{*}, \text{figure 4.14}) \\
 &\quad + D_{AND} + D_{MUX}.
 \end{aligned}$$

Even if the number of entries is 1 (which decreases the delay by the find-first-one circuit and the multiplexer) this computation cannot be done in one cycle for a stage depth δ of 5. Thus, the computation must be pipelined if the stage depth is too small.

No pipelining registers are inserted inside the last part of the computation starting at the signal req_{*} . The bound for δ given by the last part of the path is acceptable. The pipelining of the path until the signal $ready_{*}$ can be done similarly to the forwarding circuit with stalling, i.e., using two-dimensional pipelining (see section 2.6.3). In one dimension the path from the signals $\{ROB, CDB\}.*$ to the signal $ready_{*}$ is pipelined, the second pipelining dimension is the movement of the instruction through the reservation station queue.

In order to insert a pipelining register after the computation of the request signal some modifications have to be made to the reservation station entries (see figure 4.11) in order to maintain correctness. The modification to the entries is depicted in figure 4.17. The request signal is reset whenever the full bit is reset due to a dispatch or a clear. Otherwise, if the valid bit is reset in the last cycle due to dispatching the instruction or a clear, the request bit would still be active. This could lead to dispatching an

instruction twice or dispatching invalid data if the construction of the reservation station entry from figure 4.11 would not be adjusted. Note that the extra logic as depicted in figure 4.17 introduces an additional AND-gate on the path from the busses $CDB.*$ and $ROB.*$ to the full bit. Also, the clear signal is AND-ed to the full bit before the full bit is used to compute the request signal (in contrast to figure 4.11) since otherwise the delay of the path would increase by a second AND-gate.

In the memory unit the computation of the dispatch signal also depends on the AND of the signals $Entry*.write$ and $Entry*.full$ indicating valid store instructions. A register can be inserted after the computation of this AND similar to the register after the request bit. Thus, the minimum delay of the dispatch signals for the reservation stations of the memory and branch checking unit is only by D_{AND} greater than for the other reservation stations.

In order to compute the minimum delay of the inputs to the full registers and therefore the bound for the stage depth δ , the following assumptions are made: the input stall signal $stallIn$ from the FU comes directly out of a buffer circuit. Thus, the delay of this signal is D_{AND} (see section 2.5.3). A register is added directly after the request signals as in figure 4.17, leading to delay of 0 for these signals. Let i be the type of the reservation station (see table 4.1 on page 40, i.e., $i = 0$ for the memory and $i = 6$ for the branch checking reservation station). Then it must hold for a reservation station with e_{RS} entries:

$$\delta \geq \begin{cases} 3 \cdot D_{AND} & \text{if } e_{RS} = 1 \\ \max\{D_{AND}, D(\text{FFO}(e_{RS}))\} + 2 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS} > 1 \wedge i \notin \{0, 6\} \\ \max\{D_{AND}, D(\text{FFO}(e_{RS}))\} + 3 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS} > 1 \wedge i \in \{0, 6\} \end{cases} \quad (4.14)$$

Note that for a given δ this bounds the number of entries e_{RS} of the reservation stations.

Cost and Delay

To compute the number of cycles needed for dispatching instructions, two paths have to be considered. The path from the input bus $In.*$ to the register $full$ determines the minimum number of cycles that an instruction has to stay in the reservation station, if all operands of the instruction are already valid during issue. The initial value of the request signal req (i.e., the AND of the full bit and the operand valid signal) can be computed during issue without increasing the delay of the issue circuit. Thus, an issued instruction for which the operands are all valid can be dispatched in the next cycle.³

The path from the busses $ROB.*$ and $CDB.*$ to the full register determines the minimum number of cycles it takes from receiving the last operand (via $ROB.*$ or $CDB.*$) to dispatching of the instruction. In order to compute the delay of the path the boolean variable p_{req} has to be introduced, which indicates whether a pipelining register is added after the computation of the request bit. If p_{req} is one (i.e., a register is added), the delay increases by the delay of the AND which resets the request signal as in figure 4.17.

³The additional cost of $C(\text{AND-Tree}(o+1))$ is added to the cost of the reservation station

To compute the cases where p_{req} must be one the stall input $stallIn$ from the FU is again assumed to come directly out of a buffer circuit (i.e., $D(stallIn) = D_{AND}$). If $e_{RS} = 1$, the stall input $stallIn$ is at least as timing critical as the request bit. Thus, in this case p_{req} can be zero. For $e_{RS} > 1$, the delay of the find-first-one circuit is at least D_{AND} . Thus, the the path from the request signal is critical. The variable p_{req} can then be zero if the path from the signals $ready_*$ to the input of the full registers fits into one cycle. Let i be the type of the reservation station. Then p_{req} is:

$$p_{req} = \begin{cases} 0 & \text{if } \delta \geq \begin{cases} D(\text{FFO}(e_{RS})) + 3 \cdot D_{AND} + D_{MUX} & \text{if } i \neq 0 \\ D(\text{FFO}(e_{RS})) + 4 \cdot D_{AND} + D_{MUX} & \text{if } i = 0 \end{cases} \\ 1 & \text{else} \end{cases} \quad \forall e_{RS} = 1$$

Using the variable p_{req} , the delay of the path from the busses $\{ROB, CDB\}_*$ to the full register for $e_{RS} > 1$ can be computed as follows:

$$\begin{aligned} D(\{ROB, CDB\}_* \rightsquigarrow Entry_*.full') &\leq D(\text{Test}(l_{ROB})) + 2 \cdot D_{OR} \quad (Out.valid) \\ &+ D(\text{AND-TREE}(o)) \quad (ready) \\ &+ (p_{req} + 1) \cdot D_{AND} \quad (req) \\ &+ \begin{cases} 0 & \text{if } i = 6 \\ D(\text{FFO}(e_{RS})) + D_{AND} & \text{if } i = 0 \\ D(\text{FFO}(e_{RS})) & \text{else} \end{cases} \\ &+ D_{AND} \quad (disp_*) \\ &+ D_{AND} + D_{MUX}. \end{aligned}$$

If $e_{RS} = 1$ then no special computation is made for the branch checking and the memory reservation station and hence p_{req} is zero. Thus, the delay of the path is:

$$\begin{aligned} D(\{ROB, CDB\}_* \rightsquigarrow Entry_*.full') &\leq D(\text{Test}(l_{ROB})) + 2 \cdot D_{OR} \quad (Out.valid) \\ &+ D(\text{AND-TREE}(o)) \quad (ready) \\ &+ D_{AND} \quad (req) \\ &+ D_{AND} \quad (disp_*) \\ &+ D_{AND}. \end{aligned}$$

The number of cycles needed from receiving the last operand from the CDB or the RAM to dispatching the instruction c_{U2D} can be computed similarly as the forwarding circuit with stalling (see section 2.6.4):

$$c_{U2D} = \left\lceil \frac{D(\{ROB, CDB\}_* \rightsquigarrow Entry_*.full') - D_{MUX}}{\delta - D_{MUX}} \right\rceil.$$

Now consider the output bus RS_* from the reservation station to the functional unit. This bus is computed by selecting the busses $Entry_*$ using the signals $select_*$ (see figure 4.14). The delay of the busses $Entry_*$ and the signals $select_*$ is smaller than δ due to the pipelining of the path from the busses $\{ROB, CDB\}_*$ to the full bits

of the entries (see above). Due to the additional delay of the select circuit, the delay of the output bus $RS.\star$ may be greater than δ , but the select circuit can be pipelined easily since it does not contain any loops. For simplicity this is done by adding the delay of the path that does not fit into one cycle to the delay of the functional unit. Note that if the delay of the bus $RS.\star$ is smaller than δ , the delay of the functional unit can be decreased, because the functional unit can already do useful computations in the cycle, in which the instruction is dispatched.

In order to compute the delay of the bus $RS.\star$ the delay of the busses $Entry_{\star}.\star$ and the signals $select_{\star}$ after pipelining of the reservation station must be computed. Let $D'(Entry_{\star}.\star)$ and $D'(select_{\star})$ denote the delay of these signals before any pipelining registers are inserted into the path from $\{ROB, CDB\}.\star$ to the full bits of the entries. For the reservation station of the BCU ($i = 6$) the select signal only depends on the full bits of the entries. For the other reservation stations the select signals depend on the request signals $Entry_{\star}.req$. The delay of this signal can be derived from the formula above for the path from the busses $\{ROB, CDB\}.\star$ to the full bit of the entries. Thus:

$$\begin{aligned}
 D'(select_{\star}) \leq & \begin{cases} D(Entry_{\star}.req) & \text{if } i \neq 6 \\ D(Entry_{\star}.full) & \text{if } i = 6 \end{cases} \\
 & + D(\text{FFO}(e_{RS})) \\
 & \leq \begin{cases} D(\text{Test}(l_{ROB})) + 2 \cdot D_{OR} + D(\text{AND-Tree}(o)) & \text{if } i \neq 6 \\ +(p_{req} + 1) \cdot D_{AND} & \\ 0 & \text{if } i = 6 \end{cases} \\
 & + D(\text{FFO}(e_{RS})).
 \end{aligned}$$

The delay of the bus $Entry_{\star}.\star$ is dominated by the data of the operands. Thus, the delay of the bus is (see bus *Out.data* in figure 4.12).

$$D'(Entry_{\star}.\star) \leq D(\text{Test}(l_{ROB})) + \max\{D_{OR} + D_{AND}, D_{MUX}\} + D_{MUX}.$$

Due to the pipelining into c_{U2D} stages the delay of $select_{\star}$ and $Entry_{\star}.\star$ can be reduced by up to $(c_{U2D} - 1) \cdot (\delta - D_{MUX})$. Since no pipelining registers are inserted between the request signals req_{\star} and the signals $select_{\star}$, the delay of the signals $select_{\star}$ cannot get smaller than the delay from request signals. Thus, after inserting pipelining registers the delays are reduced to:

$$\begin{aligned}
 D(select_{\star}) \leq & \max\{D(\text{FFO}(e_{RS})) + \begin{cases} (1 - p_{req}) \cdot D_{AND} & \text{if } i \neq 6 \\ 0 & \text{if } i = 6 \end{cases}, \\
 & D'(select_{\star}) - (c_{U2D} - 1) \cdot (\delta - D_{MUX}), \\
 D(Entry_{\star}.\star) \leq & \max\{0, D'(Entry_{\star}.\star) - (c_{U2D} - 1) \cdot (\delta - D_{MUX})\}.
 \end{aligned}$$

If $e_{RS} = 1$ no select signals are needed to compute the output bus $RS.\star$. Thus, the delay added to the functional unit of the reservation station is:

$$D(\text{FU}(e_{RS}))^{+} \leq \begin{cases} D(Entry_{\star}.\star) - \delta & \text{if } e_{RS} = 1 \\ \max\{D(select_{\star}), D(Entry_{\star}.\star)\} - \delta & \text{if } e_{RS} > 1 \end{cases}.$$

Note that if this value is negative, the delay of the functional unit is effectively reduced, i.e., logic from the functional unit is pulled into the last cycle of dispatch.

The delay of the output stall signal *stallOut* is:

$$D(stallOut) \leq D(\text{PP-OR})(e_{RS}).$$

In order to compute the maximum number of entries e_{RS} for a given δ , the delay of the stall input *stallIn* was assume to be D_{OR} . For a given e_{RS} the delay of the stall input *stallIn* from a functional unit of type i is bounded by:

$$D(stallIn) \leq \delta - \begin{cases} 2 \cdot D_{AND} & \text{if } e_{RS} = 1 \\ (2 \cdot D_{AND} + D_{MUX}) & \text{if } e_{RS} > 1 \end{cases}. \quad (4.15)$$

The number of inputs and outputs of an entry with o operands of width w_1 to w_o and w_C control signals is $2 \cdot \sum_{j=1}^o w_j + o \cdot (l_{ROB} + 1) + 33 + l_{ROB} + w_C$ respectively $\sum_{j=1}^o w_j + 3 + w_C$. Thus, the cost of the entries increases through pipelining by:

$$C(\text{RS-Entry}(o, w_\star, w_C))^+ \leq (c_{U2D} - 1) \cdot (C_{MUX} + C_{REG}) \\ \cdot \lceil (3 \cdot \sum_{j=1}^o w_j + (o + 1) \cdot (l_{ROB} + 1) + 2 \cdot w_C + 35) / 2 \rceil.$$

The cost of a reservation station with r entries and o operands of width w_1 to w_o is:

$$C(\text{RS}(e_{RS}, o, w_\star, w_C)) \leq e_{RS} \cdot C(\text{RS-Entry}(o, w_\star, w_C)) \\ + C(\text{RS-Control}(e_{RS}, o, w_\star, w_C)) \\ + C(\text{AND-Tree}(o + 1)).$$

Let n_i be the number of functional units and e_{RS_i} the number of entries of the reservations stations of type $i \in \{0, \dots, 6\}$. Using the width of the control signals as approximated in section 4.1.6 the total cost of the dispatch phase is:

$$C(\text{Dispatch}) \leq C(\text{RS}(e_{RS_0}, 2, 32, 32, 24)) \\ + C(\text{PP-OR}(e_{RS_0} - 1)) + e_{RS_0} \cdot C_{AND} \\ + n_1 \cdot C(\text{RS}(e_{RS_1}, 2, 32, 32, 8 + l_{ROB})) \\ + n_2 \cdot C(\text{RS}(e_{RS_2}, 2, 32, 32, 8 + l_{ROB})) \\ + n_3 \cdot C(\text{RS}(e_{RS_3}, 6, 32, 32, 32, 32, 5, 2, 8 + l_{ROB})) \\ + n_4 \cdot C(\text{RS}(e_{RS_4}, 6, 32, 32, 32, 32, 5, 2, 8 + l_{ROB})) \\ + n_5 \cdot C(\text{RS}(e_{RS_5}, 6, 32, 32, 32, 32, 5, 2, 8 + l_{ROB})) \\ + C(\text{RS}(e_{RS_6}, 2, 32, 32, 96 + k_{BTB} + l_{ROB})).$$

Note the additional cost for the memory unit ($i = 0$) due to parallel prefix OR that checks for older store instructions in the queue.

| FU | CDB.hi | CDB.lo | case |
|---------|------------------------|--------------------------------|--------------------------------------|
| BCU | target PC | result | |
| ALU | result.lo | result.lo | |
| IMulDiv | result.hi | result.lo | |
| FPU | result.hi result.lo | result.lo result.lo | \overline{dbl} \overline{dbl} |
| Mem | result.lo | effective address result.lo | $dpf \vee dmal$ $(dpf \vee dmal)$ |

Table 4.3: Mapping of the FU output to the CDB

4.3 Functional Units

The processor has seven different types of functional units: memory unit, integer ALU, integer multiplier / divider, three different floating point unit types (additive, multiplicative, and misc) and a branch checking unit. To comply with the fast processor core, the fastest published additive and multiplicative floating point units are taken from [Sei99]. The delay values for these FUs are taken from [Sei03]. The miscellaneous floating point unit is not assumed to be critical and taken from [Jac02]. Delay values are from synthesis of the Verilog description [Lei02] using Synergy [Cad97].

Pipelining of the floating point and integer units is straightforward using buffer circuits if the stall path gets to long. These functional units are not described in detail here. The delay value and the computation of the stall signals for these units can be found in appendix D. The memory unit is described in chapter 5, the branch checking unit is described together with the instruction fetch in section 6.5.

Table 4.3 shows the mapping of the outputs of the FUs to the CDB. In order to allow the high part of the reservation station operands to 32 bit results, 32 bit results are written to the high and the low part of the CDB. This is only needed for FUs which may write the floating point register file (Alu, Mem, and floating point FUs).

If the memory unit returns an interrupt, the effective address is saved on the low part of the CDB. This allows to omit an additional exception data field in the ROB entries. The branch checking unit returns the target PC which is needed for interrupts of type continue (see section 4.5.3) on the high part, and the result of a branch instruction on the low part. A branch instruction produces a result if it writes to a register file entry: jump and link instructions write the address of the next instruction into the general purpose register file; return-from exception instructions write the value of the special register ESR (which is used as second operand of these instruction) into the special purpose register file.

Different units can produce different interrupts. For example only the floating point units can produce $IEEEf$ interrupts. Hence, all interrupts which cannot occur for a unit are set to zero at the output of the unit. Then all interrupt signals have defined values for all units. The outputs of the functional units have width $74 + l_{ROB}$, (the full bit, 64 data bits, 8 interrupt bits, a misprediction signal from the BCU, and l_{ROB} bits for the tag).

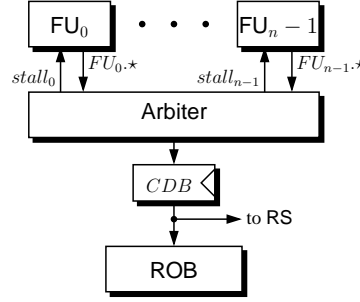


Figure 4.18: Completion phase

4.4 Completion

In the completion phase the functional units write their result to the CDB. If multiple functional units have results available, one unit is selected and may write its result in the CDB register. The other functional units are stalled. The content of the CDB register is written into the ROB and distributed to the reservation stations.

Figure 4.18 depicts an overview of the completion phase. For the completion phase the type of a functional unit is irrelevant. For simplicity the FUs are numbered from 0 to $n - 1$. The selection of the functional unit which may write to the register $CDB.*$ is done by the arbiter circuit **Arbiter**. This arbiter assigns the CDB to the functional units round robin, i.e., starting from the unit that wrote to the CDB in the last cycle, it selects the next functional unit whose output is full. If the output of no functional unit with higher index is full, the search is continued from index 0. If none of the FUs has a valid result ready the index is unchanged and the full bit of the CDB is set to 0.

Let $j^{(t)}$ denote the index of the last FU which has written to the CDB and let n be the number of functional units. The index $j^{(t+1)}$ of the next FU which may write to the CDB can be computed by the following formula:

$$j^{(t+1)} := \begin{cases} \min\{i | (FU_i.full = 1) \wedge (j^{(t)} < i < n)\} & \text{if } \bigvee_{i=j^{(t)}+1}^{n-1} FU_i.full = 1 \\ \min\{i | (FU_i.full = 1) \wedge (0 \leq i < n)\} & \text{else if } \bigvee_{i=0}^{n-1} FU_i.full = 1 \\ j^{(t)} & \text{else} \end{cases} \quad (4.16)$$

The write access to the ROB is never stalled. Hence, the register $CDB.*$ may be updated every cycle and the arbiter has no stall input.

4.4.1 Arbiter

The circuit for the arbiter is shown in figure 4.19. The sub-circuit **Ack** computes the acknowledge signals Ack which unary selects the FU that may write to the CDB. The FUs that are not selected have to be stalled. Therefore, the negation of the acknowledge signals can be used as stall input to the functional units. The circuit **Ack** also computes the full bit of the CDB.

The width of the data bus is $73 + l_{ROB}$ (64 data bits, 8 interrupt signals, the misprediction bit, and the instruction tag). Thus, cost and delay of the arbiter circuit

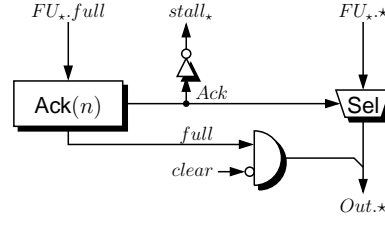


Figure 4.19: Arbiter circuit

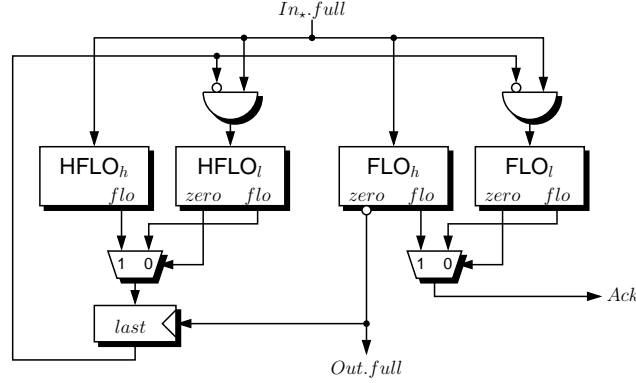


Figure 4.20: Computation of the acknowledge signals

are:

$$D(\text{Arbiter}(n)) \leq D(\text{Ack}(n)) + D(\text{Sel}(n)),$$

$$C(\text{Arbiter}(n)) \leq C(\text{Ack}(n)) + C_{AND} + (74 + l_{ROB}) \cdot C(\text{Sel}(n)).$$

Acknowledge computation

Figure 4.20 shows the implementation of the circuit **Ack** which computes the round robin acknowledge signals. It is a delay optimized version of the circuit in [Krö99] which is based on the formula (4.16). The index $j^{(t)}$ of the last functional unit which has written on the CDB is saved in the register *last* in half-unary encoding (see section 2.4), i.e., all bits of the register *last* with an index equal or lower than $j^{(t)}$ are one, the other bits are zero. Thus, the AND-gate above the rightmost find-last-one circuit FLO_l in figure 4.20 forces the full bits of those FUs to zero which have an index lower or equal to $j^{(t)}$. The circuit FLO_l thus computes the following functions:

$$FLO_l.flo := (\min\{i | (FU_i.full = 1) \wedge (j^{(t)} < i < n)\})_{un},$$

$$FLO_l.zero := \bigwedge_{i=j^{(t)}+1}^{n-1} \overline{FU_i.full}.$$

If the minimum does not exist, the zero signal is active and the output of the find-last-one circuit with unmasked inputs FLO_h is taken. If none of the full bits is active, the zero bit of the circuit FLO_h is active. Hence, the negation of this signal becomes the full bit for the CDB register.

For the register *last*, the index is needed in half-unary encoding. This is done by the find-last-one circuits that return the result in half-unary encoding $HFLO$ in the

left part of the circuit (see appendix C.1.1 for the construction of the circuit HFLO).⁴ The register *last* is only updated if a functional unit has written on the CDB, i.e., $CDB.valid = 1$.

The delay and cost of the acknowledge computation are:

$$\begin{aligned} D(\text{Ack}(n)) &\leq D_{AND} + D(\text{FLO}(n)) + D_{MUX}, \\ C(\text{Ack}(n)) &\leq 2 \cdot C(\text{FLOH}(n)) + 2 \cdot C(\text{FLO}(n)) \\ &\quad + n \cdot (2 \cdot C_{AND} + 2 \cdot C_{MUX} + C_{REG}). \end{aligned} \quad (4.17)$$

4.4.2 Pipelining

The path from the acknowledge signals to the CDB register has no loops and can therefore be pipelined easily; it is not treated in detail here. Yet the acknowledge signals *Ack* are used to compute the stall inputs of the functional units and therefore the delay must be small enough that the functional units can compute their stall signals within one cycle.

In order to compute the maximum delay allowed for the circuit $\text{Ack}(n)$, the following assumptions are made which minimize the delay of the inputs and the requirements of the acknowledge signal: the last stage of all functional units is not assumed to generate stall signals (i.e., $genStall = 0$) and are not assumed to have a buffer circuit. Thus, for all functional units i $D(FU_i.full) = 0$ holds true. The acknowledge signals are not computed using an AND-Tree. Hence, in order to be able to compute the stall signals in the functional units it must hold (see equation (2.9) in section 2.5.4):

$$\begin{aligned} \delta - D_{AND} &\geq D(\text{Ack}(n)) \\ \stackrel{(4.17)}{\Leftrightarrow} \quad \delta &\geq 2 \cdot D_{AND} + D(\text{FLO}(n)) + D_{MUX}. \end{aligned} \quad (4.18)$$

Since n must be at least 7, the proposed circuit cannot be used at a stage depth of 5. The delay of the circuit Ack can be reduced in two different ways. The arbiter can be replaced by a tree of arbiters, thus reducing the number of inputs of the single arbiters or the full outputs of the functional units can be pre-computed one cycle ahead which allows to compute the acknowledge signals in two cycles. In this thesis only the arbiter tree is presented.

Figure 4.21 depicts as an example an arbiter tree with two stages. The original arbiter with n inputs is divided into s arbiters with $t := \lceil n/s \rceil$ inputs in the first stage and one arbiter with s inputs in the second stage. All stages except the first stage of the arbiter tree have buffer circuits on their input registers in order to decouple the stall signals in the tree.

In contrary to the arbiter on the root of the tree, the arbiters in the upper nodes need to process a stall input from the lower stages. If the stall input signal is active the buffer circuit cannot accept new data. Thus, none of the acknowledge signals of the arbiter may be active. The stall signal can be incorporated in the presented arbiter circuit with only small changes that do not change the delay of the arbiter (see figure 4.22). Due to the half-unary encoding, the least significant bit of the register *last* is always 1. The

⁴The original arbiter in [Krö99] uses a parallel prefix OR over the bus *Ack* to compute the half-unary encoding which increases the delay of the arbiter circuit.

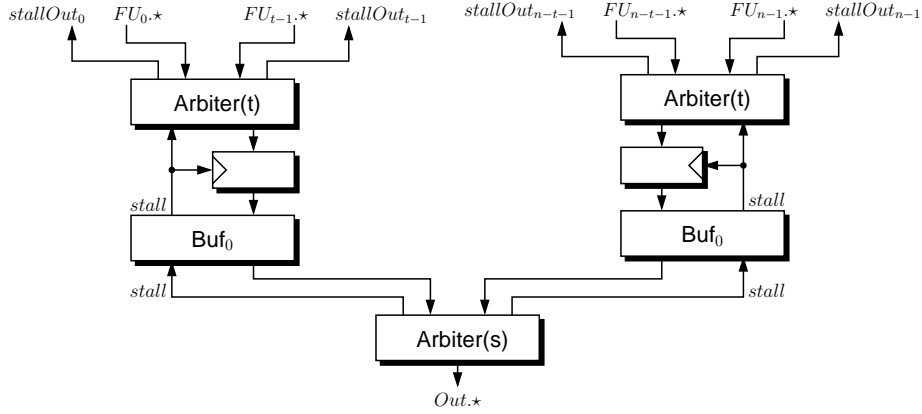


Figure 4.21: Arbiter tree

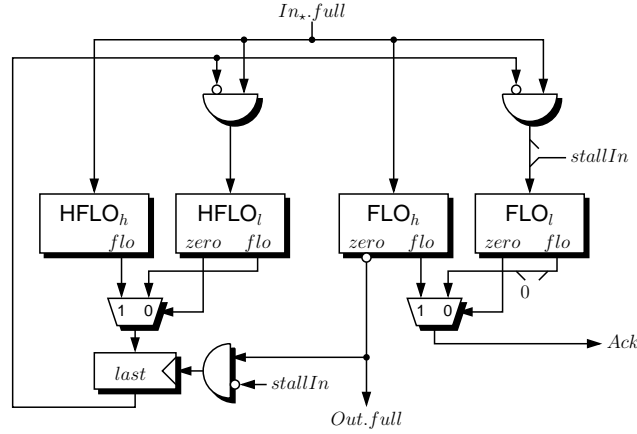


Figure 4.22: Acknowledge computation with stall input

least significant bit of the input of the rightmost find-last-one circuit FLO_l is therefore always 0 and can be ignored. Instead the stall input $stallIn$ is used as least significant bit of the input of FLO_l . Thus, if $stallIn$ is active the least significant bit of the flo output of FLO_l is active. Since this bit is replaced by a 0 in figure 4.22 and the $zero$ output of FLO_l is not active, none of the acknowledge signals is active if the stall input is active.

The register $last$ is only updated if new data is written into the buffer circuit, i.e., if the following signal is active:

$$updlast := \overline{stallIn} \wedge Out.full$$

To allow for a small stage depth, the arbitration circuit for only two inputs can be optimized using a binary encoding for the register $last$ (see figure 4.23). For an optimized delay the sub-circuit $Ack(2)$ computes different signals for stalling the inputs and for selecting between the two data ports. The stall outputs $stall_*$ are computed exploiting that the value of the stall output may be arbitrary if the corresponding input full signal is not active, since in this case the input is not stalled anyway. Also, the select signal for the data may be arbitrary if the input stall signal is active, since then the output data is not latched into the next stage anyway.

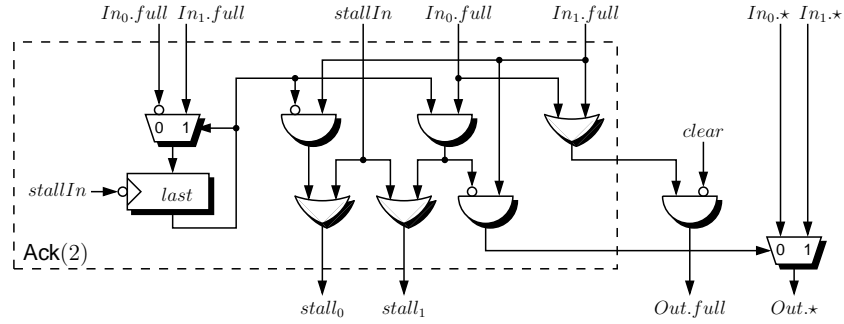


Figure 4.23: Arbiter with stall for two inputs (with sub-circuit `Ack(2)`)

The delay of the stall input of the arbiter is at most D_{AND} since it comes directly out of a buffer circuit. Thus, for the 2 input arbiter it holds (see figure 4.23):

$$\begin{aligned} D(\text{Ack}(2)) &\leq \max\{D_{MUX}, 2 \cdot D_{AND}, D_{AND} + D_{OR}\}, \\ D(\text{Arbiter}(2)) &\leq 2 \cdot D_{AND} + D_{MUX}, \\ C(\text{Arbiter}(2)) &\leq 4 \cdot D_{AND} + 3 \cdot C_{OR} + (74 + l_{ROB}) \cdot C_{MUX} + C_{REG}. \end{aligned} \quad (4.19)$$

Assume that the last stages of the functional units cannot generate a stall signal. Then for all stages of the buffer tree the delay of the input full signals is at most D_{OR} (see figure 2.4). The delay of the stall outputs of the arbiters to the preceding stage of the tree respectively to the functional unit may be at most $\delta - D_{AND}$ (see equation 2.9 in section 2.5.4). Thus, using a tree of two-port arbiters, the bound for δ from equation (4.18) can be reduced to:

$$\begin{aligned} \delta - D_{AND} &\geq D_{OR} + D(\text{Ack}(2)) \\ \stackrel{(4.19)}{\Leftrightarrow} \quad \delta &\geq D_{AND} + D_{OR} + \max\{D_{MUX}, 2 \cdot D_{AND}, D_{AND} + D_{OR}\}, \end{aligned} \quad (4.20)$$

which holds for $\delta \geq 5$. Note that this bound is independent of the number of functional units n .

4.4.3 Cost and Delay

In order to compute the number of stages of the arbiter tree of the completion phase, two variables are introduced: the variable t_L denotes the maximum number of inputs of the arbiters at the leaves of the tree, t_I denotes the maximum number of inputs of the inner nodes of the tree.

For the computation of t_L it is assumed that the last stages of the functional units do not generate a stall and do not have buffer circuits. Thus, the delay of the full signals of the inputs is zero and the delay of the outputs may be at most $\delta - D_{AND}$. Using equations (4.17) and (4.19) t_L can be computed as:

$$\begin{aligned} t_L &= \max\{t | \delta - D_{AND} \geq D(\text{Ack}(t))\} \\ &= \max\left\{t | \begin{cases} \delta \geq D(\text{FLO}(t)) + D_{MUX} + 2 \cdot D_{AND} & \text{if } t > 2 \\ \delta \geq D_{AND} + \max\{D_{MUX}, 2 \cdot D_{AND}, D_{AND} + D_{OR}\} & \text{if } t = 2 \end{cases} \right\}. \end{aligned} \quad (4.21)$$

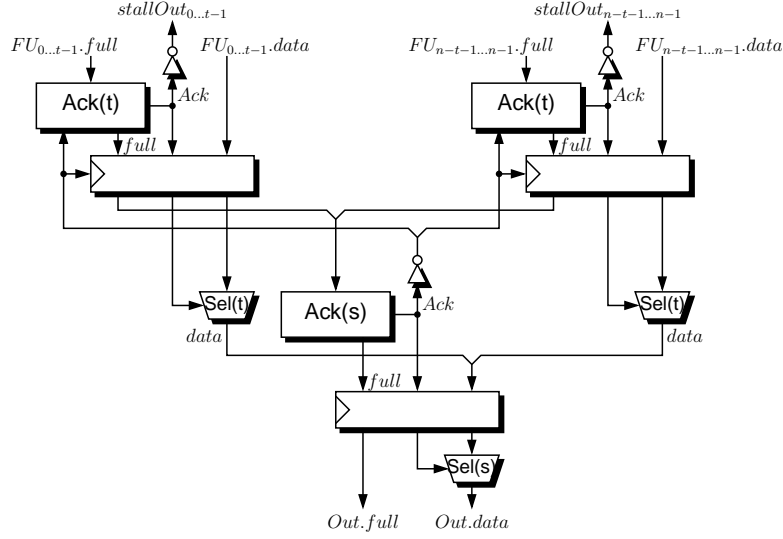


Figure 4.24: Arbiter tree with pipeline select circuits

For the inner nodes the delay of the input full signals is D_{OR} due to the buffer circuits between the stages (see section 2.5.3). Hence, t_I can be computed as:

$$\begin{aligned}
 t_I &= \max\{t | \delta - D_{AND} \geq D_{OR} + D(\text{Ack}(t))\} \\
 &= \max \left\{ t \mid \begin{cases} \delta \geq D_{OR} + D(\text{FLO}(t)) + D_{MUX} + 2 \cdot D_{AND} & \text{if } t > 2 \\ \delta \geq D_{OR} + D_{AND} & \text{if } t = 2 \end{cases} \right. \\
 &\quad \left. + \max\{D_{MUX}, 2 \cdot D_{AND}, D_{AND} + D_{OR}\} \right\}.
 \end{aligned}$$

Then the number of arbiter stages c_{AT} in the arbiter tree is:

$$c_{AT} = \lceil \log_{t_I} \lceil n/t_L \rceil \rceil.$$

The path from the acknowledge signals through the selection circuit to the data output can be easily pipelined if necessary. Note that the delay of the acknowledge computation $D(\text{Ack}(n))$ is at least as large as the delay of the select circuit $D(\text{Sel}(n))$. Thus, the output of the arbiter can be computed in at most two cycles.

In an arbiter tree, the computation of the data output of stage i of the tree can be done in parallel to the arbiter of stage $i + 1$ of the tree. See figure 4.24 for an example with 2 stages. In the figure, the select circuits **Sel(t)** compute the data for stage 1 in parallel to the acknowledge computation circuit **Ack(s)** of stage 2. Thus, only the select circuit of root of the arbiter tree needs an additional stage.

Note that performing the selection of stage i in parallel to the arbitration of stage i delays the data inputs for the section of stage $i + 1$. Thus, even if the Arbiter of stage $i + 1$ itself would fit into one cycle, the selection must then be moved to the next stage due to the delay of the data inputs. Thus, if the outputs of any arbiter in the tree is computed in two cycles, the overall number of cycles needed for the computation of the output of the arbiter tree is the number of stages of the tree plus one.

Let the boolean variable p_{AT} be one if either the arbiters at the leaves or the arbiters at the inner nodes of the tree need two cycles to compute the data outputs. For the

arbiters at the leaves of the tree the delay of the full inputs is zero. For the inner arbiters of the tree the full bits come out of a buffer circuit and therefore have delay D_{OR} . Thus:

$$p_{AT} = \begin{cases} 1 & \text{if } \max\{D(\text{Arbiter}(t_L)), D_{OR} + D(\text{Arbiter}(t_I))\} > \delta \\ 0 & \text{else} \end{cases}.$$

The total number of stages for the completion phase c_C is then:

$$c_C = c_{AT} + p_{AT}.$$

Due to the buffer circuits the delay of the stall inputs of the arbiters on the leaves of the arbiter tree is D_{AND} . Hence, the stall input is at most as critical as the full input (see figures 4.22 and 4.23). Since the full inputs of these arbiters do not come out of buffer circuits it holds:

$$\begin{aligned} D(FU_{\star} \text{stallIn}) &\leq D(\text{Ack}(t_L)) \\ &\leq \begin{cases} D_{AND} + D(\text{FLO}(\min\{t_L, n\})) + D_{MUX} & \text{if } t_L > 2 \\ \max\{D_{MUX}, 2 \cdot D_{AND}, D_{AND} + D_{OR}\} & \text{if } t_L = 2 \end{cases} \end{aligned} \quad (4.22)$$

Let n be the number of functional units. The number of inputs of the complete phase is approximately $n \cdot (74 + l_{ROB})$, the number of outputs is $74 + l_{ROB}$. Then the cost of the completion phase can be approximated by:

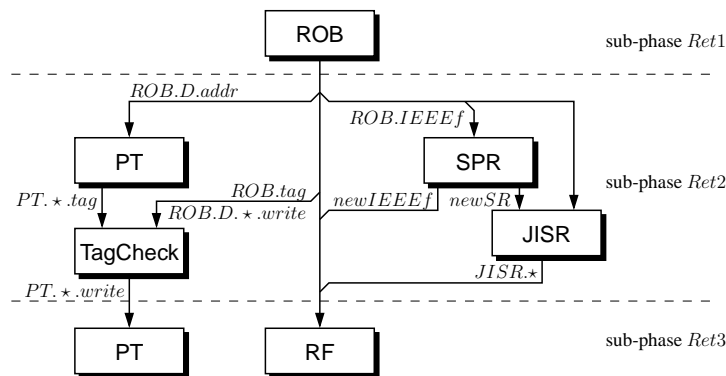
$$C(\text{Complete}) \leq \begin{cases} C(\text{Arbiter}(n)) + (n + 1) \cdot (74 + l_{ROB}) \cdot C_{REG} & \text{if } t_L \geq n \\ \begin{aligned} &\lceil n/t_L \rceil \cdot C(\text{Arbiter}(t_L)) \\ &+ (t_I^{c_{AT}-1} - 1)/(t_I - 1) \cdot C(\text{Arbiter}(t_I)) \\ &+ (c_C + 1) \cdot \lceil ((n + 1) \cdot (74 + l_{ROB}))/2 \rceil \cdot C_{REG} \end{aligned} & \text{if } t_L < n \end{cases}.$$

4.5 Retire

During the retire phase the results of the instructions are written into the register files. In order to support branch prediction and precise interrupts, the instructions are re-ordered using the reorder buffer before they write the register file. If no succeeding instruction writing the same register is processed at the time the register file is updated, the retiring instruction sets the valid bit of the producer table entry of the destination register.

4.5.1 Overview

An overview of the retire phase is depicted in figure 4.25. The retire phase is divided into three sub-phases *Ret1*, *Ret2*, and *Ret3* such that in each phase only one RAM access is made. The following sections describe the three sub-phases.

Figure 4.25: Retire phase⁵

Sub-phase *Ret1*

The first sub-phase *Ret1* reads the oldest instruction which is currently in the ROB. If this instruction in the ROB has already completed, i.e., the valid bit of the entry is set, the instruction is retired. The hardware for the sub-phase *Ret1* is presented in the ROB section 4.6.

Sub-phase *Ret2*

In the second sub-phase *Ret2* the producer table is checked for whether no succeeding instruction currently being processed write to the same destination register. Only then the valid bit of the producer table entry of the destination register may be set when the destination register is written. Otherwise the producer table entry must stay unchanged. In order to check whether the instruction being retired is the last one to write a register, the sub-phase *Ret2* reads the producer table PT for the destination register entry of the retiring and then compares the tag with the tag of the instruction in the circuit TagCheck. Since every instruction writes its tag to the producer table entry of its destination register during decode, no other instructions currently processed writes its result to the same register if and only if the tags match. If the tags don't match the write signal for the producer table is disabled.

In parallel to this check the sub-phase *Ret2* computes all remaining signals which are needed for retiring the instruction. In particular, these are the new value for the floating point flag register *IEEEf* and the interrupt bus *JISR.**. For this the sub-phase *Ret2* accesses the special purpose register which delivers the value of the special register *IEEEf* and *SR* for the time the instruction enters the sub-phase *Ret3* (see section 4.7.4). The register *SR* is then used to compute the interrupt bus.

Sub-phase *Ret3*

In the sub-phase *Ret3* the producer tables and the register files are updated. If an interrupt or a branch misprediction has occurred, the processor is flushed by activating the signal *clear*. The signal *clear* resets the full bits of all stages as well as the

⁵The producer tables and the SPR are accessed twice during retire (the SPR is accessed the second time as part of the register files). Both RAMs are depicted twice in order to emphasize the independent accesses.

producer tables and the ROB. Since retire is done in order this leaves the processor in a consistent state. The computation of the clear signal depends on the handling of branch misprediction and is therefore presented in more detail in the instruction fetch chapter 6. The sub-phase *Ret3* only consists of these RAM accesses and is presented in the sections for the register file and the producer table (4.7 and 4.8).

4.5.2 Tag Check

The circuit *TagCheck* checks for all register files $\mathfrak{R} \in \{\text{GPR}, \text{FPR}, \text{SPR}\}$ if the tag of the destination register $PT.\mathfrak{R}.tag$ of the instruction equals the tag $ROB.tag$ of the instruction read from the ROB. If the tags are equal and the write signal for the register file type $ROB.D.\mathfrak{R}.write$ is active, the valid bit of the destination register entry must be set by writing to the producer table. Thus, the write signals for the producer tables are computed as follows:

$$D.PT.\mathfrak{R}.write := D.\mathfrak{R}.write \wedge (ROB.tag = PT.\mathfrak{R}.tag).$$

Let I be an instruction in the retire sub-phase *Ret2*. The output $PT.\mathfrak{R}.tag$ of the read access to the producer table for I must contain the value of the entry at the time the instruction I enters the sub-phase *Ret3*. Otherwise it could happen that a new instruction I_n which updates the producer table during its decode phase is not recognized. The updated value of I_n could be overwritten by the instruction I in sub-phase *Ret3* which may lead to data inconsistencies. Hence, all write accesses to the PT during decode that start before I enters the sub-phase *Ret3* must be forwarded to the read access in the retire sub-phase *Ret2*. The forwarding of the producer table described in detail in section 4.8.

The cost and delay of the circuit *TagCheck* are:

$$\begin{aligned} C(\text{TagCheck}) &\leq 3 \cdot C(\text{EQ}(l_{ROB} + 1)), \\ D(\text{TagCheck}) &\leq D(\text{EQ}(l_{ROB} + 1)). \end{aligned}$$

4.5.3 Interrupt Handling

The circuit *JISR* computes all interrupt related signals. If an interrupt occurs, the processor is flushed and the instruction fetch is restarted at the start of the interrupt service routine (ISR). The service routine then executes code to react on the interrupt (it “handles” the interrupt). The last instruction of the ISR is always a return-from-exception (rfe) instruction which returns to the code that caused the interrupt. In order to allow precise interrupt handling some registers of the SPR defining address and type of the interrupt must be updated when an interrupt occurs. A detailed description of the interrupt handling including a correctness proof can be found in [MP00].

The supported interrupts are shown in table 4.4 ordered by priority. If two different interrupts occur for one instruction, the interrupt with higher priority (lower index) is handled first. The internal interrupts (priority 1 to 12) are detected in the phases instruction fetch, decode, and execute. The corresponding signals are collected and saved in the ROB. The external interrupts are assigned to the first instruction which enters the retire sub-phase *Ret2* after the interrupt occurred. As in [MP00], the external

| name | signal | priority | type | maskable | external | |
|-----------------------|------------------------|----------|----------|----------|----------|----|
| reset | <i>reset</i> | 0 | abort | no | yes | |
| illegal instruction | <i>ill</i> | 1 | | | repeat | no |
| misaligned access | <i>mal</i> | 2 | | | | |
| page fault IM | <i>Ip_f</i> | 3 | | | | |
| page fault DM | <i>Dp_f</i> | 4 | | | | |
| trap | <i>trap</i> | 5 | continue | yes | | |
| FXU overflow | <i>ov_f</i> | 6 | | | | |
| FPU overflow | <i>fOV_F</i> | 7 | | | | |
| FPU underflow | <i>fUN_F</i> | 8 | | | | |
| FPU inexact result | <i>fIN_X</i> | 9 | | | | |
| FPU divide by zero | <i>fDB_Z</i> | 10 | | | | |
| FPU invalid operation | <i>fIN_V</i> | 11 | | | | |
| FPU unimplemented | <i>uFOP</i> | 12 | | | | |
| external I/O | <i>ex_j</i> | 12+j | | no | yes | |
| | | | | yes | | |

Table 4.4: Interrupts

interrupts signals are required to remain active until the processor is flushed. The internal and external interrupts are combined in the bus $CA[31 : 0]$ according to their priority:

$$CA[i] := \begin{cases} pup & \text{if } i = 0 \\ ROB.ill & \text{if } i = 1 \\ ROB.dmal \vee ROB.imal & \text{if } i = 2 \\ ROB.Ipj & \text{if } i = 3 \\ ROB.Dpj & \text{if } i = 4 \\ ROB.trap & \text{if } i = 5 \\ ROB.ovf & \text{if } i = 6 \\ ROB.IEEEf[i - 7] & \text{if } 7 \leq i \leq 12 \\ ex[i - 13] & \text{if } i \geq 13 \end{cases}$$

Misaligned memory accesses can occur during instruction fetch (*imal*) or during data memory accesses (*dmal*). Both interrupts are combined to the misaligned interrupt signal $CA[2]$.

The interrupts 6 to 11 and 13 to 31 can be masked by the mask register SR . These interrupts are ignored if the corresponding bit of the register SR is not set. The masked interrupt bus MCA is defined as:

$$MCA[i] := \begin{cases} CA[i] \wedge SR[i] & \text{if } 6 \leq i \leq 11 \vee 13 \leq i \\ CA[i] & \text{if } i \leq 5 \vee i = 12 \end{cases}$$

The signal $jisr$ indicates that a non-masked interrupt has occurred:

$$jisr := ROB.full \wedge \bigvee_{i=0}^{31} MCA[i]$$

An interrupt can be of one of the types *abort*, *repeat*, or *continue*. If an abort interrupt occurs, the processor is restarted. If the interrupt is of type *repeat*, the instruction that caused the interrupt has to be repeated after the interrupt has been handled by the ISR. If the interrupt is of type *continue*, the processor continues at the succeeding instruction after the execution of the ISR. Since the content of the register files is irrelevant if the interrupt is of type *abort*, the processor core distinguishes only between the types *repeat* and *continue*:

$$repeat := MCA[3] \vee MCA[4] = CA[3] \vee CA[4].$$

Repeat interrupts have higher priority than continue interrupts and are not maskable. Thus, the interrupt with highest priority cannot be of type *continue* if *repeat* is active.

The PC of the instruction which must be executed after the interrupt has been handled is stored in the special register *ePC*. For abort interrupts the value written into this register may be arbitrary. If the interrupt is of type *repeat*, this is the PC of the instruction that caused the interrupt. If the interrupt is of type *continue*, this is the PC of the next instruction. The PC of the next instruction depends on the type of the instruction for which the interrupt occurred. If the instruction is a branch instruction (indicated by the signal *branch* in the ROB entry), the branch target has been computed in the BCU, which delivers the target on the high part of the result bus (see table 4.3). Otherwise the PC of the next instruction is the PC of the current instruction plus 4. The special register *ePC* is updated using the bus *ePC* which therefore is defined by:

$$ePC := \begin{cases} ROB.data.hi & \text{if } \overline{repeat} \wedge ROB.branch \\ ROB.PC + 4 & \text{if } \overline{repeat} \wedge \overline{ROB.branch} \\ ROB.PC & \text{if } repeat \end{cases}.$$

If the interrupt is caused by a trap instruction, the register *eData* must be updated with the immediate constant of the instruction. If the interrupt is caused by a page fault or a misaligned memory access the register *eData* must be updated with the effective address of the memory access. In any case the value that has to be written into the register *eData* can be found in the low part of the result bus *ROB.data* as explained in the following.⁶

For instruction memory interrupts, the decode circuit sets the immediate constant to the PC of the instruction, i.e., the effective address of the memory access (see section 4.1.3). Thus for both trap instructions and instruction memory interrupts the register *eData* must be set to the value of the immediate constant. In both cases the instruction causing the interrupt uses the integer ALU which then copies the immediate constant to the low part of the result bus. If a data memory interrupt occurs, the memory unit also stores the effective address on the low part of the result (see table 4.3). The special register *eData* is updated using the bus *eData*. Thus:

$$eData := ROB.data.lo.$$

⁶Thus, the ROB entries of the DLX_{π+} do not need an extra 32 bit field for the exception data as the designs proposed in [Krö99] and [Hil00].

Before the processor jumps into the ISR, all instructions preceding the instruction that caused the interrupt must have updated the register files. This is guaranteed as retiring is done in order. The instruction causing the interrupt may only update the register files if it is not of type repeat. Thus, the write signal for a register file $\mathfrak{R} \in \{\text{GPR}, \text{FPR}, \text{SPR}\}$ may not be active for interrupts of type repeat:

$$D.RF.\mathfrak{R}.write := D.\mathfrak{R}.write \wedge \overline{repeat}.$$

The bus $JISR.\star$ updates the instruction fetch unit. The full bit must be active if an interrupt occurred. It forces the instruction fetch unit to continue the instruction fetch at the start of the ISR, i.e., at address $JISR.sisr$. This bus is set to the constant SISR (start of the interrupt service routine):

$$\begin{aligned} JISR.full &:= jisr, \\ JISR.sisr &:= SISR. \end{aligned}$$

Stalling

The last stage of the retire sub-phase $Ret2$ has to be stalled if an interrupt occurs and the instruction fetch unit cannot accept new data, which is indicated by the signal $\overline{IF.lastcycle}$ (see section 6.1.2). If the retire sub-phase $Ret2$ is divided into multiple stages the stalling only affects the last stage. The other stages are never stalled. Since the interrupt flushes the whole processor core anyway, inconsistent data for the succeeding instructions do not affect the correctness. Let c_{Ret2} be the number of stages of the retire sub-phase $Ret2$. Then:

$$\begin{aligned} Ret2.stall^i &:= 0 \text{ for } i \in \{0, \dots, c_{Ret2} - 2\}, \\ Ret2.stall^{c_{Ret2}-1} &:= JISR.full \wedge \overline{IF.lastcycle}. \end{aligned}$$

Cost and Delay

The cost and delay of the circuit JISR are:

$$\begin{aligned} C(JISR) &\leq C(\text{OR-Tree}(32)) + C(\text{INC}(32)) + 32 \cdot C(\text{Sel}(3)) + 30 \cdot C_{AND} + C_{OR}, \\ D(JISR) &\leq \max\{D(\text{INC}(32)), 2 \cdot D_{AND} + D(\text{OR-Tree}(32))\}. \end{aligned}$$

4.5.4 Cost and Delay

Sub-phase $Ret1$

For performance measurements the number of cycles needed to read out the ROB in the sub-phase $Ret1$ has no impact. The minimum number of cycles between the completion and the retiring of an instruction is defined by the number of cycles c_{C2R} it takes to forward the data on the CDB to the output of the ROB in the read access in sub-phase $Ret1$. The value of c_{C2R} is computed in the ROB section 4.6. The retire sub-phase $Ret1$ needs no additional hardware apart of the ROB.

Sub-phase *Ret2*

The total delay and the number of stages needed for the the sub-phase *Ret2* are:

$$D(Ret2) \leq \max\{D(PT.\star.tag) + D(\text{TagCheck}), D(SPR.newSR) + D(\text{JISR}), \\ D(SPR.newIEEEf)\}, \\ c_{Ret2} \leq \lceil D(Ret2)/\delta \rceil.$$

The inputs to the retire phase *Ret2* are the outputs of ROB and producer table and the external interrupts. Thus, the number of inputs is 140. The outputs of the retire phase *Ret2* are the interrupt bus *JISR.★*, the destination register bus *D.★*, and the interrupt signals for the special purpose register file. The number of output bits is 264. The cost of the sub-phase is:

$$C(Ret2) \leq C(\text{TagCheck}) + C(\text{JISR}) + 5 \cdot C_{OR} + c_{Ret2} \cdot 182 \cdot C_{REG}.$$

Sub-phase *Ret3*

The number of stages of the retire sub-phase *Ret3* has no impact on the performance of the processor. The cost of the sub-phase *Ret3* is the cost of the input registers:

$$C(Ret3) \leq 264 \cdot C_{REG}.$$

4.6 Reorder Buffer Environment

The reorder buffer is a queue used to rearrange instructions into program order before they are retired. This is needed for precise interrupt handling [SP85]. New entries in the ROB are allocated during decode. When an instruction completes, the valid bit of the entry is set. If the oldest instruction in the ROB is valid, it is retired.

4.6.1 Overview

The ROB is implemented as a RAM block with head and tail pointer. The head pointer addresses the oldest entry, the tail pointer points to the entry which should be filled next with a new instruction. When an instruction retires, the head pointer is incremented; when a new entry is allocated, the tail pointer is incremented. If the ROB is full no new instructions are decoded (see section 4.1.7). Thus a new entries are only allocated, if the ROB is not full.

The ROB is read in two different contexts and written in two further contexts. In order to distinguish these four contexts a name is introduced for every context. The four different accesses to the ROB are listed in the following:

Allocation-context: During the decode sub-phase *D2* (see section 4.1.1) a ROB entry is allocated for the new instruction by writing to the ROB. The address of the entry which is allocated is given by the tail pointer. This access resets the valid bit of the entry to indicate that the instruction has not yet completed and writes the information for the instruction which are known during decode (e.g., the address of the destination register) into the ROB.

| group | name | width | purpose |
|---------|----------------------|-------|-----------------------------------|
| valid | <i>valid</i> | 1 | valid signal for entry |
| dataLow | <i>data</i> [31 : 0] | 32 | lower 32 bit of result |
| dataHi | <i>data</i> [31 : 0] | 32 | upper 32 bit of result |
| onIssue | <i>ill</i> | 1 | illegal instruction |
| | <i>imal</i> | 1 | misaligned IMem access |
| | <i>ipf</i> | 1 | IMem page fault |
| | <i>trap</i> | 1 | trap instruction |
| | <i>uFOP</i> | 1 | unimplemented FP instruction |
| | <i>D.addr</i> | 5 | destination address |
| | <i>D.dbl</i> | 1 | double precision result |
| | <i>D.GPR.write</i> | 1 | GPR destination |
| | <i>D.FPR.write</i> | 1 | FPR destination |
| | <i>D.SPR.write</i> | 1 | SPR destination |
| onCompl | <i>branch</i> | 1 | branch instruction |
| | <i>writeIEEEf</i> | 1 | instruction writes IEEEf register |
| | <i>PC</i> | 32 | instruction PC |
| | <i>dmal</i> | 1 | misaligned DMem access |
| | <i>dpf</i> | 1 | DMem page fault |
| | <i>ovf</i> | 1 | ALU overflow |
| | <i>IEEEf</i> | 5 | IEEE flags |
| | <i>mp</i> | 1 | misprediction |

Table 4.5: Components of an ROB entry

Operand-read-context: In parallel to the access-context the ROB is read for each of the (up to six) operands of the new instruction. If the entry read for the operand has already completed, the result can be forwarded to the reservation station.

Completion-context: During completion the result of the instruction is written into the ROB. This access also sets the valid bit of the ROB entry.

Retiring-context: In the retire sub-phase *Ret1* the ROB entry addressed by the head pointer is read in order to retire the oldest instruction. If the valid bit of the read instruction is set, the instruction is retired.

Table 4.5 lists the fields of the ROB entries. Not all fields are used in every context. The components which are used in the same contexts are combined into the following groups as proposed in [Hil00]:

valid: This group contains the valid bit of the entry. This bit indicates that the instruction which is saved in this entry has already completed and hence the ROB entry contains the valid result of this instruction. It is accessed in every context.

dataHi, dataLo: The two groups dataHi and dataLo contain the high respectively low part of the result. Both groups are read in the operand-read context and the retire-context and written in the completion-context. Note that data needed by the high

| group | ports | | | | width |
|---------|------------|--------------|------------|----------|-------|
| | allocation | operand-read | completion | retiring | |
| valid | 1W | 6R | 1W | 1R | 1 |
| dataLo | | 3R | 1W | 1R | 32 |
| dataHi | | 3R | 1W | 1R | 32 |
| onIssue | 1W | | | 1R | 48 |
| onCompl | | | 1W | 1R | 9 |

Table 4.6: Data width and number of ports of the ROB groups

parts of the operands OP_1 and OP_2 can only be found in the group dataHi (see section 4.1.2). The low part of the operands OP_1 and OP_2 can only be found in the group dataLo. The operands OP_3 and OP_4 can only be written by 32 bit results and can therefore be found in both the groups dataHi and dataLo (see section 4.3). In order to distribute the read ports evenly for the operand OP_3 the group dataLo is read and for the operand OP_4 the group dataHi is read. Therefore, only three read ports are needed for the operand-read context.

onIssue: All information about the instruction which do not change after the decode phase are combined in the group onIssue. This comprises the interrupt conditions which are detected during instruction fetch and decode, the information about the destination register, the PC of the instruction, the *branch* bit indicating a branch instruction (i.e., a branch or a jump), and the bit *writeIEEEf* indicating a moveI2S instruction which writes the special register *IEEEf*. The group onIssue is written in the allocation-context and read in the retire-context.

onCompl: This group contains the interrupt conditions which are detected during execute and the branch misprediction signal. Since these informations are not needed by succeeding instructions this group is not accessed in the operand-read context. Thus, the group is only accessed in the completion- and retiring-context.

Table 4.6 shows the width and the number of ports needed by the groups of the ROB. The valid group needs seven read ports. To reduce the number of read ports the three copies of the RAM block for the valid group are used with the same write ports but less read ports. The first two RAM blocks with 3 read ports each correspond to the RAM blocks of the groups dataLo and dataHi and are accessed in the same contexts. The last RAM block has only one read port and is accessed in the retire context. In order for all sub-groups to have the same content, every sub-group has the same two write ports. Since the valid group consists of only one bit, the additional cost for the RAM blocks is not significant.

4.6.2 Pipelining of the Retiring-Context

Pipelining of the ROB access in the retiring-context must be done differently to all other RAM accesses as the decision which address has to be read next depends on the last read result.

Let c_{Ret1} be the number of cycles needed for a read access to the ROB in the retire-context. In [Krö99] the read access for the oldest instruction is restarted if the valid bit of the oldest instruction is not set. Then no instruction could be retired for the next $c_{Ret1} - 1$ cycles. This degrades the performance if c_{Ret1} is larger than one. In order to avoid a restart of the read access, the read access of the oldest instruction I is stalled in the last stage of the ROB if the I has not completed yet. Upon completion of I the result is forwarded to this read access. The forwarding also sets the valid bit of the data read from the ROB and the instruction can retire.

Since a read access for the ROB takes c_{Ret1} cycles, the read access for the second-oldest instruction must already have been started, when the oldest instruction is retired. Otherwise instructions could only be retired every c_{Ret1} cycles. Let t be the tag of the oldest instruction in the ROB. In order to be able to retire an instruction every cycle, the read access for the entry $t + 1$ must be in the second-last stage of the read access, the access for entry $t + 2$ in the third last and so on. If the processor is flushed, the oldest instruction in the ROB will have the tag 0. Thus, upon activation of the clear signal the pipeline of the read access is set up such that the last stage contains a read access for the entry zero, the second last stage a read access for the entry one, and so on. Thus, the first stage of the read access in the retire context must contain a read access for the entry $c_{Ret} - 1$. The address of the first stage of the read access is determined by the head pointer. Thus, the head pointer does not point to the oldest instruction, but to the $c_{Ret1} - 1$ -oldest instruction. If the oldest instruction is retired, all read accesses move to the next stage and the head pointer is incremented. Hence, if t is the tag of the oldest instruction in the ROB, the stages 0 to $c_{Ret1} - 1$ of the read access in the retire-context always contain read accesses to the entries $t + c_{Ret1} - 1$ to t .

Note that the read accesses in the retire context are started speculatively, i.e., a read access can be started even before the entry has been allocated for an instruction. Thus, the RAM will not return any valid data for the read access. In this case all information of the instruction must be returned by means of forwarding.

4.6.3 Forwarding

In this section the forwarding of the write accesses in the allocation- and completion-contexts to the read accesses in the operand-read- and retiring-contexts is described. Every of the four possible combinations is discussed separately.

Allocation-Context to Operand-Read-Context

Only the valid bit of the ROB entries are written in the allocation-context and read in the operand-read context. The allocation-context disables the valid bit. Hence, forwarding of this bit would mean that the entry of an instructions that has already retired is invalidated. Yet, the register file and producer table environments use the fact that the results of instructions that have been retired can still be read out of the ROB. This allows to omit forwarding in the register file and producer table environment from the retire phase to the decode phase (see section 4.7.1). Hence, no forwarding is done from the allocation-context to the read-context.

The read accesses in the operand-read-context can be stalled since it has to wait until the instruction is issued (see section 4.1.7). The write accesses in the allocation-

context is not stalled. Thus, a write accesses can overtake a read accesses. If the read access uses the same address as the write access, the read access would return the data written by the write access. Thus, even if no forwarding is done, the read access could return data written by write accesses that have not been started before. In order to prevent this it must be guaranteed that the write accesses that can overtake a read do not use the same address as the read access.

The accesses in the allocation and operand-read-context for an instruction are both started when the instruction enters the decode sub-phase $D2$. The read access is stalled in the first stage of $D2$ if the output stall signal of the issue circuit is active. The read access must not finish before an instruction is issued (see section 4.1.7). If issuing can be done in one cycle the read access does not have to be stalled, once it enters the second stage of $D2$. If issuing takes multiple cycles, the last stage of the ROB read access is stalled if the control signal *issued* is not active. Thus, in this case all stages of the ROB access can be stalled.

The write access in the allocation-context is never stalled. In fact if the instruction is stalled in the first stage of the decode sub-phase $D2$ multiple write accesses will be started for the instruction. Hence, if issuing can be done in one cycle, the read access in the operand-read-context of an instruction I can be overtaken by the write access in the allocation-context of I itself. Once the read access is in the second stage no more write accesses can overtake the read. Thus, in this case it must be guaranteed that the tags of the operands of I returned by the producer table may not have the same value as the tag of I .

If issuing takes multiple cycles, a read access can be stalled in every stage of the access. Hence, a read access in the operand-read-context of an instruction I can be overtaken by the write accesses in the allocation context of all instructions that can be in the decode sub-phase $D2$ at the same time as I . Let c_{D2} be the number of cycles of the decode sub-phase $D2$ and let t be the tag of I . Since decode is done in order, the writes that can overtake the read access in the operand-read-context of I have the tags $t + c_{D2} - 1$ to t (modulo the size of the ROB L_{ROB}). Thus it must be guaranteed, that the instruction I does not depend on any instruction with these tags.

Completion-Context to Operand-Read-Context

If not all operands of an instruction I are valid, the results of all instructions that have not updated the register files must be forwarded to the instruction I until all operands are valid. The read access to the ROB RAM in the operand-read-context of instruction I returns the results of all instructions which have already completed at the time the read access is started. As soon as the instruction I is in a reservation station, the forwarding is done by the reservation station by snooping on the CDB. Thus, the forwarding circuit for the ROB read access in the completion-context must take all instructions into account that complete in the window from the start of the read access of the instruction I to the arrival of I in the reservation station.

If issue is done in one cycle, the window consists of exactly the one cycle in which the instruction is issued. The forwarding of this cycle is done by a forwarding tree with one input. If the issuing takes multiple cycles, the CDB has to be forwarded until the signal *issued* is active (see section 4.1.7). Since the read access can be stalled if issuing takes multiple cycles, the forwarding is done by means of a forwarding circuit

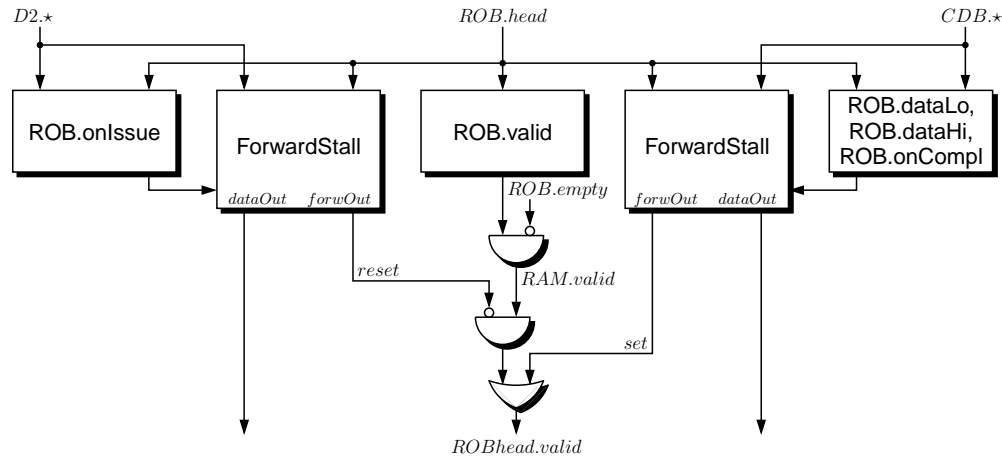


Figure 4.26: Forwarding for the ROB access in the retiring-context

with stalling.

Allocation-Context to Retiring-Context

The read access in the retiring-context is done speculatively. Thus the read access in the retiring-context for an instruction I can have been started even before an entry was allocated for I in the allocation-context. Therefore, the allocate context must be forwarded to the retiring-context in order to read, e.g., the destination address. Also the forwarding from the allocation-context must reset the valid bit of the entry. Otherwise if the the read access in the retiring-context is started before the write in the allocation-context, the read could return a spurious valid signal. As the read access in the retiring-context can be stalled, the forwarding is done using a forwarding circuit with stalling.

Completion-Context to Retiring-Context

Due to the speculative read, the read access in the retiring-context for an instruction I can have been started before the instruction writes its result to the ROB in the completion context. Thus, the write access in the completion context is also forwarded to the retiring-context by means of a forwarding circuit with stalling.

4.6.4 Implementation of Forwarding

Figure 4.26 depicts an overview of the ROB forwarding in the retiring-context. Note that all groups of the ROB RAM except the valid group are only written in either the allocation- or the completion-context. Thus, for each of these groups only one forwarding circuit is needed. The valid bit is reset in the allocation-context and set in the completion-context and therefore depends on both forwarding circuits.

The output of the ROB is forced to zero when the ROB is empty (indicated by the signal **ROB.empty**, computed by the ROB control), resulting in the signal **RAM.valid**. This is done to prevent spurious valids. This output of the RAM is combined with the signals **reset** and **set** generated from the outputs **forwOut** of the forwarding circuits

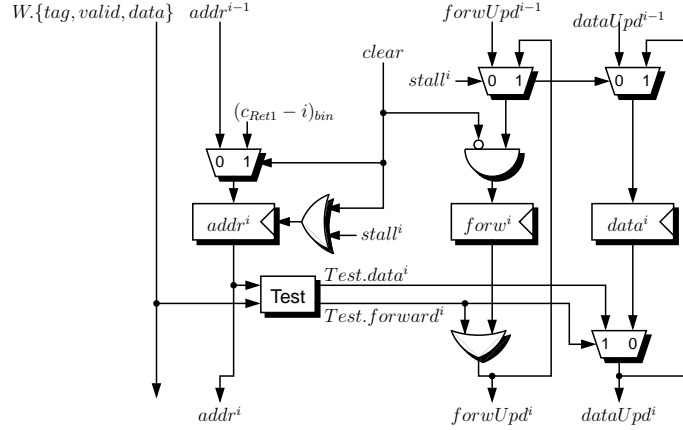


Figure 4.27: Stage i of the forwarding circuit for the ROB

for the allocate- respectively completion-context. These signals indicate that the last stage of the corresponding forwarding circuit contains data to be forwarded.

The signal *reset* is active if the allocation of the entry for an instruction I is done after the read access in the retire context for I has been started. Since the allocation resets the valid bit the signal *reset* forces the signal *RAM.valid* to zero. The signal *set* is active if the instruction stored in the ROB entry has completed since the RAM access has been started. This signal forces the signal *RAM.valid* to one, resulting in the signal *ROBhead.valid*. The signal *set* has higher priority than the signal *reset* as the write in the completion-context for an instruction is always started after the write in the allocation-context. The signal *ROBhead.valid* indicates whether the oldest instruction in the ROB is valid. It is used as full input of the retire sub-phase *Ret2*.

Note that upon activation of the clear signal the ROB access in the retiring-context is not started from the beginning, but the whole pipeline is filled with the read requests for the ROB entries $c_{Ret1} - 1$ to 0 (see section 4.6.2). The valid output *ROBhead.valid* of the RAM access in the retire-context depends on the forward signals of the forward circuits, hence these signals have to be reset upon clear. Figure 4.27 depicts a modified stage of the forwarding circuit for the read access in the retiring context (see figure 2.10 on page 21 for the unmodified stage). If the signal *clear* is active, the stage resets the bit *forw* and sets the address of the access to $c_{Ret1} - i$.

4.6.5 Control

Stall signals

The write accesses in the allocation- and the completion-context are never stalled. The stalling of the ROB read accesses in the operand-read context has been discussed in section 4.1.7. The read-access in the retiring-context has to be stalled if the valid bit *ROBhead.valid* (see figure 4.26) of the oldest instruction (which is in the last stage of this read-access) is not active:

$$Ret1.stallIn := \overline{ROBhead.valid}.$$

If *ROBhead.valid* is not active, the oldest instruction is stalled in the last stage of the read access in the retiring-context until the instruction completes and the forward-

ing circuit for the completion-context forces the signal $ROBhead.valid$ to one, as depicted in figure 4.26.

This stall signal is used in a forwarding circuit with stalling. Thus it must hold (see equation (2.11) on page 21):

$$D(ROBhead.valid) := D(Ret1.stallIn) \leq \delta - D_{MUX}. \quad (4.23)$$

The signal $ROBhead.valid$ is computed using the outputs $forwOut$ of forwarding circuits with stalling (see figure 4.26). The outputs are based on the updated forward bits of the last stage of forwarding circuit (see figure 4.27). Thus, the delay of the output $forwOut$ must hold:

$$\begin{aligned} D(forwOut) + D_{OR} + D_{AND} &\leq D(ROBhead.valid) \stackrel{(4.23)}{\leq} \delta - D_{MUX} \\ \Leftrightarrow D(forwOut) &\leq \delta - (D_{MUX} + D_{OR} + D_{AND}). \end{aligned} \quad (4.24)$$

If a pipelining register is added after the test circuit in the stages of the forwarding circuit (see figure 4.27), the outputs $forwUpd^*$ of the stages have delay D_{OR} . Hence the output $forwOut$ of the forwarding circuit has delay D_{OR} and the equation holds true for $\delta = 5$.

Head and tail pointer

The head and the tail pointer of the ROB are implemented using two counters in the circuit **HeadTail** (see figure 4.28). The head pointer addresses the oldest instruction in the ROB. When this instruction is retired (i.e., $ROBhead.valid = 1$), the head pointer is incremented. The tail pointer addresses the next free entry of the ROB. If a new instruction enters the core, the value of the tail pointer is assigned as tag to the instruction and the tail pointer is incremented. A new instruction enters the core if the first stage of the decode sub-phase $D1$ is full and not stalled. The incrementation of head and tail pointer is controlled by the signals $headce$ and $tailce$:

$$\begin{aligned} headce &:= ROBhead.valid, \\ tailce &:= D1.full^0 \wedge \overline{D1.stall^0} \\ &= D1.full^0 \wedge \overline{D1.full^0 \wedge (D1.stallIn^0 \vee D1.genStall^0)} \\ &= D1.full^0 \wedge \overline{(D1.stallIn^0 \vee D1.genStall^0)}. \end{aligned}$$

If the signal $clear$ is active the head and the tail pointer are reset. The tail pointer is set to zero. Due to the pipelining of the ROB accesses the head pointer is set to $c_{Ret1} - 1$, where c_{Ret1} is the number of stages of the retire sub-phase $Ret1$ (see section 4.6.2).

The delay of the computation of the new values for the head and tail pointer (see the figure 4.28) is $D(\text{Inc}(l_{ROB})) + D_{MUX}$. If this cannot be computed in one cycle, the counter can be pipelined easily. Upon clear of the counters the pipelining registers must be setup such that the counters increment correctly from 0 respectively $c_{Ret1} - 1$. The cost of the circuit **HeadTail** can be estimated as:

$$C(\text{HeadTail}) \leq 2 \cdot (C(\text{Inc}(l_{ROB})) + C_{OR} + l_{ROB} \cdot (C_{MUX} + C_{REG})) + C_{AND}.$$

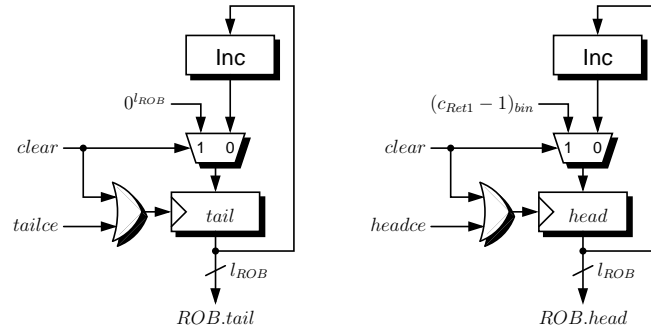


Figure 4.28: Head and tail pointer computation

Full and empty bit

The ROB control computes two signals $ROB.full$ and $ROB.empty$ indicating that the ROB is full respectively empty. The signal $ROB.full$ is used as stall signal in the decode phase, since new instruction may only enter the decode phase if the ROB is not full. If no instruction is in the ROB, the output of the ROB is invalid. In order to prevent an invalid instruction from retiring, the valid output of the ROB is ignored if the signal $ROB.empty$ is active as depicted in figure 4.26.

The computation of the full and empty bits is based on a circuit which counts the number of valid instructions in the ROB as proposed in [Lei99]. Due to the pipelining of the decode and retire phases two separate counters have to be used for the computation of the full and the empty signal.

The ROB full signal is used to guarantee tag-uniqueness, i.e., that no two instructions with the same tag are active at the same time. Therefore, the counter for the ROB full signal must take all instructions into account which have already entered the decode sub-phase $D1$ but have not yet written the register files, i.e., entered the retire sub-phase $Ret3$. These instructions are called *active* instructions.

The ROB empty signal is used to invalidate the ROB output if the ROB does not contain any valid instruction. Therefore, the counter for the ROB empty signal must only count the instructions which really are in the ROB, i.e., the instructions for which the write access in the allocation-context during decode has been started and the read access in the retiring-context has not yet finished. The write access in the allocate context is started in decode sub-phase $D2$. The read access in the retiring-context is finished if an instruction enters the retire sub-phase $Ret2$.

In [Lei99] decoding or retiring an instruction only takes one cycle. Thus, instructions allocate a ROB entry in the same cycle they enter the decode phase and write the register files in the same cycle they are read out of the ROB. Therefore, the same counter can be used for the full and the empty bit in [Lei99].

Figure 4.29 depicts the circuit FullEmpty with the counters $fcnt$ and $ecnt$ for the computation of the ROB full signal $ROB.full$ and ROB empty signal $ROB.empty$.

Instructions get active when they enter the decode sub-phase $D1$. This is indicated by the signal $tailce$. Instructions get inactive when they enter the last retire sub-phase $Ret3$. Since this sub-phase is never stalled, this is indicated by the full bit of the input registers $Ret3.full^0$. Hence, the signals $tailce$ and $Ret3.full^0$ can be used to increment respectively decrement the counter $fcnt$.

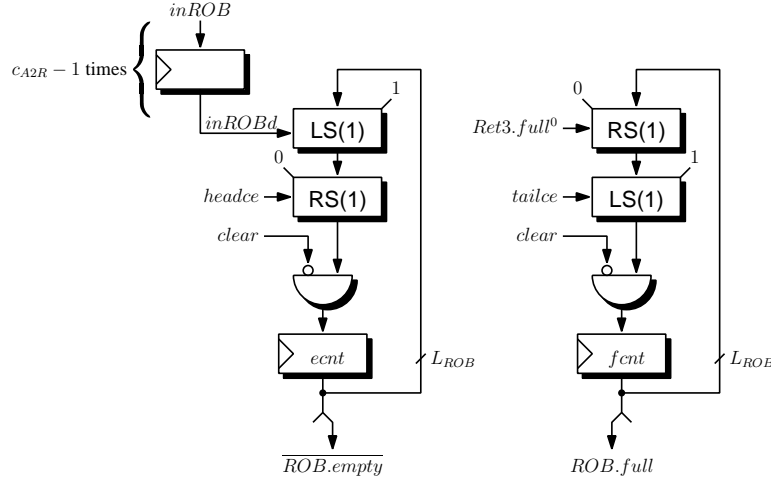


Figure 4.29: Computation of the full and empty bit

An instruction is read out of the ROB if it is the oldest instruction in the ROB and already has completed. In this case the instruction enters the retire sub-phase *Ret2*. This is indicated by the signal *headce* which thus determines whether the counter *ecnt* shall be decremented. A new write access in the allocate-context is started if an instruction is in the first stage of the decode sub-phase *D2* and this stage is not stalled:

$$inROB := D2.full^0 \wedge \overline{D2.stall^0}$$

Thus, the signals *inROB* can be used to increment the counter *ecnt* with the following small restriction: Let c_{A2R} be the number of cycles needed for forwarding from the allocation-context to the retiring-context. Thus, it takes c_{A2R} cycles to clear the valid output of the ROB *ROBhead.valid* using forwarding (by the signal *reset*, see figure 4.26). Then the signal *inROB* must be delayed $c_{A2R} - 1$ cycles and the delayed version *inROBd* must be used to increment the counter *ecnt* in figure 4.29. Otherwise the empty signal might become inactive before the write in the allocation-context is forwarded to the retiring-context and has reset the valid output *RAM.valid* of the ROB. The valid bit stored in the ROB RAM could still be active from the last instruction that used the entry. Thus the signal *ROBhead.valid* could be active which would lead to a spurious retirement of an invalid ROB entry.

To reduce the delay of the counters *fcnt* and *ecnt*, half-unary counters are used. This allows to increment the counter with a 1-bit left-shifter *LS(1)* and to decrement with a 1-bit right-shifter *RS(1)* (see the figure 4.29). Since the last bit of the half-unary encoding is always zero these bits are ignored for the counters, i.e., a 0 is represented by all bits being zero. Both counters can be cleared using the signal *clear*.

The ROB empty signal *ROB.empty* must be inactive if the value represented by *ecnt* is at least one. Thus:

$$\begin{aligned}
 ROB.empty &:= \begin{cases} 0 & \text{if } \langle ecnt \rangle_{hun} \geq 1 \\ 1 & \text{else} \end{cases} \\
 &= \overline{ecnt[0]}.
 \end{aligned} \tag{4.25}$$

The ROB full signal must be computed differently depending on whether issuing can be done in one cycle. Let c_I be the number of cycles needed for issuing an instruction and let c_{D2} be the number of cycles needed for the read access to the ROB in the operand-read-context. As described in section 4.6.3 up to c_{D2} writes in the allocation-context can overtake a read of the ROB in the operand-read context if issuing cannot be done in one cycle (i.e., $c_I > 1$). Otherwise only the write access in the allocation-context of an instruction can overtake its own read access in the operand-read-context.

If issuing can be done in one cycle, the ROB full signal is set to one, if $L_{ROB} - 1$ instructions are active, otherwise it is already set to one if $L_{ROB} - c_{D2}$ instructions are active. This is required by the correctness proofs of the forwarding for register file and producer table. It follows:

$$\begin{aligned} ROB.full &:= \begin{cases} 1 & \text{if } c_I = 1 \wedge \langle fcnt \rangle_{hun} \geq L_{ROB} - 1 \\ 1 & \text{if } c_I > 1 \wedge \langle fcnt \rangle_{hun} \geq L_{ROB} - c_{D2} \\ 0 & \text{else} \end{cases} \\ &= \begin{cases} fcnt[L_{ROB} - 2] & \text{if } c_I = 1 \\ fcnt[L_{ROB} - 1 - c_{D2}] & \text{if } c_I > 1 \end{cases} \end{aligned} \quad (4.26)$$

Note that the new value of the registers $ecnt$ and $fcnt$ must be computed in one cycles based on the old value. Thus

$$\begin{aligned} \delta &\geq D(\text{LS}(1)) + D(\text{RS}(1)) + D_{AND} \\ &= 2 \cdot D_{MUX} + D_{AND} \end{aligned} \quad (4.27)$$

which holds for $\delta \geq 5$. The cost of the circuit **FullEmpty** is:

$$\begin{aligned} C(\text{FullEmpty}) &\leq 2 \cdot L_{ROB} \cdot (2 \cdot C_{MUX} + C_{AND} + C_{REG}) \\ &\quad + (c_{A2R} - 1) \cdot C_{REG} + C_{AND}. \end{aligned}$$

Check for oldest instruction

Instructions reading the special register $IEEEf$ must wait at the stage c_{RF} of the decode sub-phase $D1$ until all preceding instructions have retired, which is indicated by the signal $allRet$ (see section 4.1.7). This is necessary as instructions write the register $IEEEf$ implicitly, i.e., forwarding is not done using the Tomasulo algorithm.

Figure 4.30 shows the computation of the signal $allRet$. Let $D1.tag^{c_{RF}}$ be the tag of the waiting instruction. The instruction is the oldest instruction in the ROB if the tag of the instruction in the last stage of the read access in the retiring-context of the ROB $ROB.tag$ matches $D1.tag^{c_{RF}}$. No instruction is in the retire sub-phase $Ret2$ if all full bits of $Ret2$ are zero. This is indicated by the signal $Ret2.clean$. All instructions preceding the waiting instructions have retired if the tags match and $Ret2.clean$ is active.

The cost of the computation of the signal $allRet$ is:

$$C(\text{allRet}) \leq C(\text{EQ}(l_{ROB})) + c_{Ret2} \cdot C_{AND} + 2 \cdot C_{REG}.$$

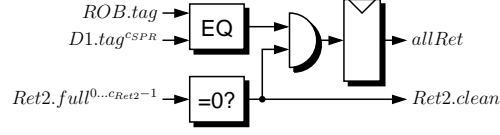


Figure 4.30: Computation of the signal *allRet*

4.6.6 Correctness

The goal of this thesis is not to formally prove the correctness of the $DLX_{\pi+}$. Yet “paper-and-pencil” proofs are given for the parts of the design of the $DLX_{\pi+}$ that differ from a Tomasulo DLX with a simple pipeline (of which the correctness is formally proven, e.g., in [Krö01]) and that are not obviously correct. These are especially the different computation of the ROB full and empty signals and the forwarding. Note that some of the lemmas proven here are needed for the correctness proofs of the forwarding of register files and producer tables in sections 4.7.1 and 4.8.1.

The following definitions are used in the theorems and lemmas below.

Definition 4.1. An instruction is called “active” if it has entered the decode sub-phase D1, but has not written the register files (i.e., has not entered the retire sub-phase Ret3).

An instruction is called “in the ROB” if for this instruction the write access in the allocate-context has been started, but the instruction has not yet entered the retire sub-phase Ret2.

Let c_{A2R} be the number of cycles needed for forwarding from the allocation- to the retiring-context. An instruction is called “visible in the ROB” if the ROB write access in the allocation context has been started at least $c_{A2R} - 1$ cycles before, but the instruction has not yet entered the retire sub-phase Ret2.

The ROB is called empty if no instruction is in the ROB.

Note that from the time instruction is visible in the ROB, all read accesses in the retiring-context that finish, will take the write access in the allocation-context into account due to forwarding. This cannot be guaranteed for instruction that are “in the ROB” but not “visible in the ROB”. Also note that every instruction that is “visible in the ROB” is also “in the ROB” and every instruction that is “in the ROB” is also “active”. If decoding and retiring is done in one cycle and $c_{A2R} = 1$ (as, e.g., in [Lei99]) all three terms define the same set of instructions.

Lemma 4.2. The number of active instructions is equal to $\langle fcnt, 1 \rangle_{hun}$. The number of instructions that are visible in the ROB is equal to $\langle ecnt, 1 \rangle_{hun}$.

Proof. The lemma can be proven by induction on the number of cycles since the last cycle in which the clear signal was active.

Induction base: The clear signal is active, hence the core is cleared and no instruction is active or visible in the ROB at the end of the cycle. The counters *fcnt* and *ecnt* are set to $0^{L_{ROB}}$ due to the clear signal (see figure 4.29), which proves the induction base.

Induction step: Assume the clear signal is not active and the claim holds for the previous cycle. The counter *fcnt* is increased if a new instruction becomes active and

it is decreased if an instruction retires. The signal $inROBd$ is active if the allocate access for an instruction has been started exactly c_{A2R} cycles ago. This signal increments the counter $ecnt$. The signal $headce$ which is active if an instruction enters the retire sub-phase $Ret2$ decrements the counter $ecnt$. Thus, it remains to show that the counters do not overflow or underflow to prove the induction step.

If the signal $ROB.full$ is active, the first stage of the decode sub-phase $D1$ is stalled (see section 4.1.7). Thus, no instruction can become active, the counter $fcnt$ cannot overflow. As every instruction that is visible in the ROB is also active the value of $ecnt$ cannot be larger than the value of $fcnt$ and hence $ecnt$ cannot overflow, too. If $ROB.empty$ is active, no instructions can enter the sub-phase $Ret2$. Thus, the counter $ecnt$ cannot underflow. Since an instruction cannot be visible in the ROB if it is not active, the value of $fcnt$ must be larger than the value of $ecnt$ and hence $fcnt$ cannot underflow. \square

Corollary 4.3. *Let c_{D2} be the number of cycles needed for the ROB access in the operand-read-context. If issuing is done in one cycle, at most L_{ROB} instructions are active at any time. If issuing is done in multiple cycles, at most $L_{ROB} - c_{D2} + 1$ instructions are active at any time.*

Proof. If issuing is done in one cycle and $L_{ROB} - 1$ instructions are active, the signal $ROB.full := fcnt[L_{ROB} - 2]$ is active (see equation (4.26)). This stalls the first stage of decode sub-phase $D1$. Thus, including the instruction in the first stage of decode sub-phase $D1$, at most L_{ROB} instructions can be active. If issuing is done in multiple cycles the statement can be proven analogously from equation (4.26). \square

Corollary 4.4. *No two instructions with the same tag can be active at the same time.*

Proof. Assume two instructions I, I' with the same tag are active. The l_{ROB} bit wide tail counter is incremented whenever an instruction becomes active. Thus, at least $L_{ROB} - 1 = 2^{l_{ROB}} - 1$ instructions must have become active between the two instructions I, I' . The instructions I and I' are active and therefore have not yet retired. Since retire is done in order it follows that the other instructions have not yet retired, too. Thus, at least $L_{ROB} + 1$ instructions must be active which contradicts corollary 4.3. \square

Corollary 4.5. *If the signal $ROB.empty$ is not active, the ROB is not empty. Let in this case I be the oldest instruction in the ROB. Then the allocate access for I has been started at least $c_{A2R} - 1$ cycles before.*

Proof. If $ROB.empty$ is not active, the signal $ecnt[0]$ is one (see equation (4.25)). Thus, $\langle ecnt, 1 \rangle_{hun} \geq 1$. It follows from lemma 4.2 that there is at least one instruction which is visible in the ROB. This instruction is therefore in the ROB and the ROB is not empty.

Let I be the oldest instruction in the ROB. Since allocation is done in order, the instructions that are visible in the ROB cannot have started the ROB access in the allocation-context before I . Hence, I must also be visible in the ROB. \square

Lemma 4.6. *Let c_{D2} be the number of cycles needed for the ROB access in the operand-read-context and let I be an active instruction with tag t . If issuing is done in*

one cycle, no older active instruction can have the tag t . If issuing is done in multiple cycles no active instruction can have one of the tags t to $t + c_{D2} - 1$ (modulo the size of the ROB L_{ROB}).

Proof. Let I_0 be an active instruction with tag t_0 and let I_1 be an instruction older than I_0 with tag t_1 . Let d be such that $t_0 = t_1 + d \pmod{L_{ROB}}$ for $d \in \{1, \dots, L_{ROB}\}$. The tail pointer assigning tags to the instruction is incremented for every instruction becoming active. Since decoding and retiring are done in order, for every $j \in \{0, \dots, d\}$ an instructions with tag $t := t_1 + j \pmod{L_{ROB}}$ must be active. Hence, at least those $d + 1$ instructions are active.

If issuing is done in one cycle, it follows from corollary 4.3 that $d < L_{ROB}$ and hence $t_0 \neq t_1$. If issuing is done in multiple cycles, it follows from the corollary that $d < L_{ROB} - c_{D2} + 1$. Hence, the instruction t_1 cannot have one of the tags t_0 to $t_0 + c_{D2} - 1$ (modulo L_{ROB}). \square

The lemma guarantees that decoding is stopped before any ROB entries that are read in the operand-read-context are invalidated by the allocation-accesses of succeeding accesses.

Theorem 4.7. *Let L_{ROB} be the number of entries of the ROB and let c_{Ret1} be the number of cycles needed for the ROB read access in the retiring-context. If $L_{ROB} > c_{Ret1}$ and the signal $ROBhead.valid$ is active, then the ROB is not empty and the instruction in the last stage of the read access in the retire-phase has already completed.*

Proof. Assume the signal $ROBhead.valid$ is active. For this either the signal set or the output of the RAM must be active (see figure 4.26).

Case 1, the signals set is not active. Then the output of the ROB valid RAM must be one and the signals $reset$ and $ROB.empty$ must be inactive. Thus, according to corollary 4.5 the ROB is not empty. Let I be the oldest instruction in the ROB. It follows from corollary 4.5 that a write access in the allocation-context for I has been started at least $c_{A2R} - 1$ cycles before. Since $reset$ is not active, this write access must have been started before the read access in the retiring-context has been started. Therefore, the RAM block returns the correct value for the valid bit of instruction I . Since the output of the RAM is active, I must have already completed.

Case 2, the signal set is active. Let t be the address (i.e., the tag) of the last stage of the ROB access in the retiring-context. If the signal set is active, an instruction I with tag t has completed after the ROB access in the retiring-context to the entry with address t has been started. An instruction can only complete if it has been issued before. The write access in the allocation-context is started when an instruction is issued. Thus, the instruction I must already have started the write access in the allocation-context. This proves the first part of I is in the ROB from definition 4.1.

In order to prove that I is in the ROB it remains to show that instruction I has not yet retired. From the correctness of the Tomasulo algorithm follows that no instruction can complete twice. Since set is active the instruction I cannot have already completed before the read access in the retiring-context with tag t has been started. Thus, the instruction I cannot have retired without setting the signal set of an read access in the retiring-context as proven in case 1. From $L_{ROB} > c_{Ret1}$ it follows that no two stages with the same address can exist in the retire access to the ROB. Thus, the result of the

instruction I can only have been forwarded to the read which is now in the last stage of the ROB access in the retiring-context. Hence, the instruction I cannot have retired yet.

Since the instruction I has not yet retired, it must be in the ROB, the ROB is not empty. It follows that the instruction in the last stage of the retiring-context is the instruction I , which has already completed. \square

Note that the requirement $L_{ROB} > c_{Ret1}$ of the theorem holds true for reasonable values of L_{ROB} .

Theorem 4.8. *If the ROB is not empty, the signal $ROBhead.valid$ gets active eventually.*

Proof. Let I be the oldest instruction in the ROB. Since the read access in the retiring context is stalled if $ROBhead.valid$ is inactive, the instruction I remains the oldest instruction unit $ROBhead.valid$ gets active.

The liveness of the Tomasulo algorithm guarantees that the instruction I completes eventually (as proved, e.g., in [Krö01]). If the read access in the retiring-context for I has been started before I completes, the signal set gets active eventually which forces $ROBhead.valid$ to one.

If the read access in the retiring-context for I has been started after the instruction completes, the ROB RAM for the valid bit returns a one. The write access in the allocation-context is started in parallel to the issue of an instruction. Thus, it must have been started before the instruction completes. According to corollary 4.4 no other instruction with the same tag as I can be active. Therefore, no write access in the allocate context can have been started to the entry of instruction I and hence $reset$ cannot be active. Since the allocation access has already been started at most after c_{A2R} cycles the signal $ROB.empty$ gets inactive and hence $ROBhead.valid$ gets active. \square

Cost

The total cost of the ROB control is (including the gates to compute the valid output $ROBhead.valid$):

$$C(\text{ROB-Control}) \leq C(\text{HeadTail}) + C(\text{FullEmpty}) + C(\text{allRet}) \\ + 2 \cdot C_{AND} + C_{OR}.$$

4.6.7 Delay Optimizations

The critical signals of the ROB control are the clock enables for the head and tail pointer $headce$ and $tailce$. In the described implementation the signals $headce$ and $tailce$ may have a delay of at most $\delta - (D_{MUX} + D_{AND})$ as they control the counters $ecnt$ and $fcnt$

The signal $headce$ is derived directly from the signal $ROBhead.valid$ which has at most the delay $\delta - D_{MUX}$ (see equation (4.23) on page 83). In the computation of the ROB empty signal $ROB.empty$ in figure 4.29 the AND gate for clearing the counter can be moved above the right shifter controlled by $headce$ (see the left part

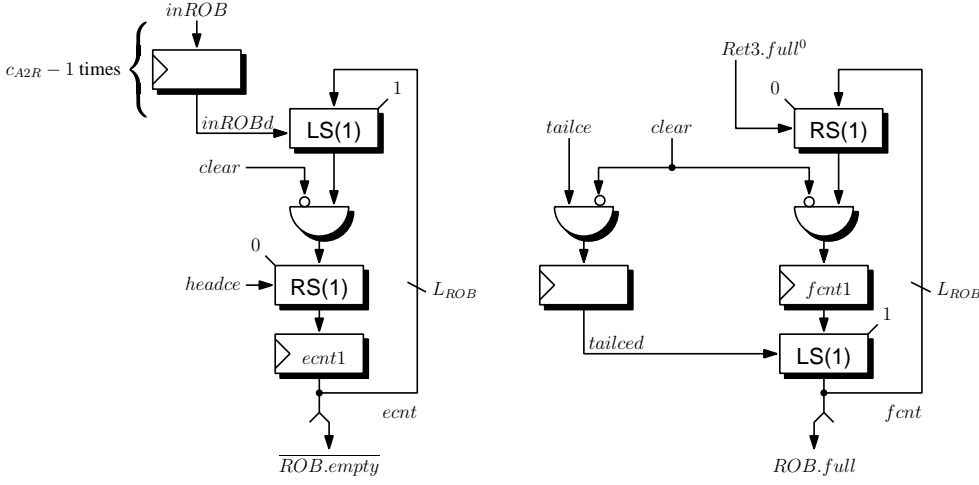


Figure 4.31: Optimized Computation of the full and empty bit

of figure 4.31). Even if *headce* is active while clearing the counter, the value of the counter will still be set to all zeros as the right-shift only increases the number of zeros. Then the signal *headce* may have a delay of $\delta - D_{MUX}$.

The signal *tailce* is derived from the signal $D1.stallIn^0 \vee D1.genStall^0$ (see section 4.1.8) which is assumed to have a delay of at most δ . On the other hand the delay of the signal *ROB.full* was assumed to be D_{MUX} in the computation of the stall signals for the decode phase. This allows moving the left shifter in the computation for the signal *ROB.full* in figure 4.29 below the register and using a delayed version *tailced* of the signal *tailce* to control the shifter (see the right part of figure 4.31). If the counter is reset, the left-shift behind the register is prevented by forcing the signal *tailced* to zero on clear. The AND gate does not increase the delay of the signal as

$$\begin{aligned} tailce \wedge \overline{clear} &= D1.full^0 \wedge \overline{(D1.stallIn^0 \vee D1.genStall^0)} \wedge \overline{clear} \\ &= \neg(D1.full^0 \vee D1.stallIn^0 \vee D1.genStall^0 \vee clear) \end{aligned}$$

and the OR-Tree can be balanced such that the delay is the same as the delay of the signal $D1.stallIn^0 \vee D1.genStall^0$ (see figure 4.9 on page 50). Hence, in the circuit in figure 4.31 only bounds δ by the loops through *ecnt* and *fcnt*:

$$\delta \geq 2 \cdot D_{MUX} + D_{AND},$$

which holds true for $\delta = 5$.

The signal *headce* and *tailce* are also used to control the clocking of the head and tail pointers in figure 4.28. Due to the clear logic the delay of the signals may be at most $\delta - D_{OR}$, which is already achieved for *headce*. However, to achieve this requirement for the signal *tailce* the logic for clearing the tail counter must be changed (see figure 4.32). The circuit in the figure forces the output of the register *tail* to zero from the cycle after the clear signal has been active to the cycle where the first instruction enters the decode sub-phase *D1*. As a side-effect the modification also forces the output to zero, whenever the input register of the decode sub-phase *D1* is not full, but in this case the tail pointer may have arbitrary value.

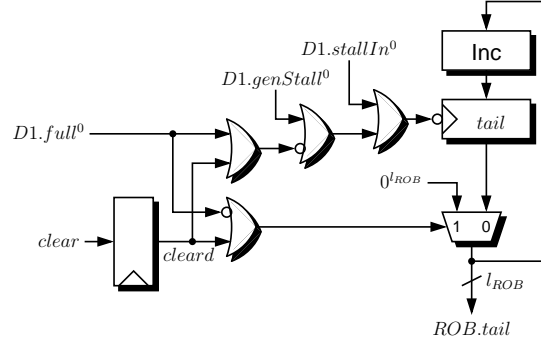


Figure 4.32: Optimized clearing of the tail counter

The circuit delays the signal *clear* is delayed by one cycle. This allows to optimize the computation of the clock enable signal of the register *tail*, since the signal *clear* resets the full bits of the decode stage, the ROB full signal *ROB.full*, and the signal *IR.haltdec* (see section 6.4). Thus, when the delayed clear signal *cleard* is active, the signals *D1.stallIn*⁰ and *D1.genStall*⁰ are inactive, i.e.:

$$cleard = 1 \Rightarrow D1.stallIn^0 \vee D1.genStall^0 = 0. \quad (4.28)$$

Therefore, the signal *cleard* \vee *tailce* used to reset the output of the tail counter can be simplified as:

$$\begin{aligned} cleard \vee tailce &= cleard \vee (D1.full^0 \wedge \overline{(D1.stallIn^0 \vee D1.genStall^0)}) \\ &= (cleard \vee D1.full^0) \wedge \overline{(cleard \vee (D1.stallIn^0 \vee D1.genStall^0))} \\ &\stackrel{(4.28)}{=} (cleard \vee D1.full^0) \wedge \overline{(D1.stallIn^0 \vee D1.genStall^0)} \\ &= \neg((cleard \vee D1.full^0) \vee D1.stallIn^0 \vee D1.genStall^0). \end{aligned}$$

The OR-Tree can again be balanced such that the delay of the signal *cleard* \vee *tailce* has the same delay as the signal *D1.stallIn*⁰ \vee *D1.genStall*⁰ which is at most δ .

In order to force the signal *ROB.tail* to zero until the first instruction enters the decode sub-phase *D1* the register *tail* is also cleared if the signal *D1.full* is not active. Since the stall signal for the first stage cannot get active until the first instruction enters the decode sub-phase *D1* the stall signal must not be taken into account.

Assume the number of cycles c_I needed for issuing is greater than one. Then the write access in the completion-context must be forwarded to the read access in the operand-read-context with a forwarding circuit with stalling. This requires that the stall signals *D2.stall*^{*i*} of all stages *i* of the read access in the operand-read-context have at most delay $\delta - D_{MUX}$ (see equation (2.11) on page 21). If the read access takes c_{D2} cycles and the stall signals are computed the usual way, it must hold for the stall signal for the second stage:

$$\begin{aligned} \delta - D_{MUX} &\geq D(D2.stall^1) \\ &= D(ROB.stallOut) \stackrel{(4.10)}{=} D(\text{AND-Tree}(c_{D2} + 1)) \end{aligned}$$

For $l_{ROB} \geq 6$ and $\delta = 5$ it holds $c_{D2} > 7$. Then, the equation does not hold.

To reduce the delay of the stall signal Instead pipeline bubbles are only removed in the last seven stages of the ROB read access, i.e., only the full bits of the last seven stages are taken into account for computing the stall signals as proposed in the first part of section 2.5.3. Thus, for all other stages the stall signal is computed as:

$$stall = \bigwedge_{i=c_{D2}-6}^{c_{D2}} D2.full^i \wedge D2.issued^{c_{D2}}$$

The additional cost for the delay optimizations are:

$$\begin{aligned} C(\text{HeadTail})^+ &\leq 2 \cdot C_{OR} + C_{REG}, \\ C(\text{FullEmpty})^+ &\leq C_{AND} + C_{REG}. \end{aligned}$$

4.6.8 Cost and Delay

The delay of the ROB access in the operand-read context determines the delay of the decode sub-phase $D2$. The delay of the ROB access in the retire context determines the delay of the retire sub-phase $Ret1$. The delay of the ROB accesses are:

$$\begin{aligned} D(D2) &\leq \max\{D(\text{RAM}(L_{ROB}, 1, 3, 2)), D(\text{RAM}(L_{ROB}, 32, 4, 1))\} + D_{MUX}, \\ D(Ret1) &\leq \max\{D(\text{RAM}(L_{ROB}, 1, 1, 2)) + 2 \cdot D_{AND} + D_{OR}, \\ &\quad \max\{D(\text{RAM}(L_{ROB}, 32, 4, 1)), D(\text{RAM}(L_{ROB}, 48, 1, 1)), \\ &\quad D(\text{RAM}(L_{ROB}, 9, 1, 1))\} \\ &\quad + D_{MUX}\}. \end{aligned}$$

Let c_{D2} and c_{Ret1} be the number stages of the two ROB accesses respectively the corresponding sub-phases. Define c_{ROB} to be the maximum number of cycles needed for any of the two ROB read accesses:

$$\begin{aligned} c_{D2} &= \lceil D(D2)/\delta \rceil, \\ c_{Ret1} &= \lceil D(Ret1)/\delta \rceil, \\ c_{ROB} &= \max\{c_{D2}, c_{Ret1}\}. \end{aligned}$$

The number of stages for the retire sub-phase $Ret1$ is only needed for cost computations. The minimum number of cycles between the complete and the retire of an instruction is bounded by the number of cycles needed for forwarding with stalling c_{C2R} :

$$c_{C2R} = \lceil (D(\text{Test}(l_{ROB})) + 2 \cdot D_{OR} + D_{AND} - D_{MUX}) / (\delta - D_{MUX}) \rceil.$$

The number of cycles c_{A2R} needed to forward from the write accesses in the allocation-context is computed analogously, i.e., $c_{A2R} = c_{C2R}$.

The cost of the ROB RAM blocks can be estimated as:

$$\begin{aligned} C(\text{ROB-RAM}) &\leq C(\text{RAM}(L_{ROB}, 1, 1, 2, c_{ROB})) \\ &\quad + 2 \cdot C(\text{RAM}(L_{ROB}, 1, 3, 2, c_{ROB})) \\ &\quad + 2 \cdot C(\text{RAM}(L_{ROB}, 32, 4, 1, c_{ROB})) \\ &\quad + C(\text{RAM}(L_{ROB}, 48, 1, 1, c_{ROB})) \\ &\quad + C(\text{RAM}(L_{ROB}, 9, 1, 1, c_{ROB})). \end{aligned}$$

For the retire context two forwarding circuits with stalling are needed. One forwarding circuit forwards the valid and the onIssue group from the allocate context (width 48), the other circuit forwards the valid, dataLo, dataHi, and the onCompl group from the complete context (width 74). Let c_I be the number of stages of the issue circuit. If $c_I = 1$ the forward circuit in the forward context consists of a Forwarding Tree with one leaf for each operand. If $c_I > 1$ a forward circuit with stalling is needed for each operand.

Let $c_{FS}(l_{ROB})$ be the number of cycles needed for forwarding with stalling (with l_{ROB} address bits). Then the cost of the forwarding circuits of the ROB can be estimated as:

$$\begin{aligned}
 C(\text{ROB-Forward}) \leq & C(\text{ForwardStall}(l_{ROB}, 48, c_{Ret1}, c_{FS})) \\
 & + C(\text{ForwardStall}(l_{ROB}, 74, c_{Ret1}, c_{FS})) \\
 & + \begin{cases} 4 \cdot C(\text{Forward-Tree}(l_{ROB}, 32, 1, c_{D2})) \\ \quad + C(\text{Forward-Tree}(l_{ROB}, 5, 1, c_{D2})) & \text{if } c_I = 1 \\ \quad + C(\text{Forward-Tree}(l_{ROB}, 2, 1, c_{D2})) \\ 4 \cdot C(\text{ForwardStall}(l_{ROB}, 32, c_{D2}, c_{FS})) \\ \quad + C(\text{ForwardStall}(l_{ROB}, 5, c_{D2}, c_{FS})) & \text{if } c_I > 1 \\ \quad + C(\text{ForwardStall}(l_{ROB}, 2, c_{D2}, c_{FS})) \end{cases}
 \end{aligned}$$

The total cost of the ROB environment is:

$$C(\text{ROB}) \leq C(\text{ROB-RAM}) + C(\text{ROB-Control}) + C(\text{ROB-Forward}).$$

4.7 Register File Environment

The processor has three different types of register files: the general purpose register file GPR, the floating point register file FPR, and the special purpose register file SPR. All register files have one write port used in the retire sub-phase *Ret3* to store the result and multiple read ports used in the decode sub-phase *D1* to read the operands. The SPR has two additional read ports and one additional write port. The additional read ports are used in the retire sub-phase *Ret2*. These read ports always read the registers *SR* and *IEEEf* and therefore have constant address busses. The additional write port is used in sub-phase *Ret3* to store the new value of the register *IEEEf* computed in the sub-phase *Ret2* from the old value and the floating point exceptions of the instruction. The standard port for storing the result cannot be used to write the register *IEEEf* as floating point compare instructions may write two special registers, namely *FCC* for the comparison result and *IEEEf* for the exceptions.

The addresses of the read accesses during decode sub-phase *D1* are computed in the register file environments from the instruction register $IR[31 : 0]$. The encodings of the addresses in the instruction words is listed in table A.8 in the appendix A. The address and write signals used for the write access are computed in the retire sub-phase *Ret2*. They are combined in the bus *Ret.D.★*.

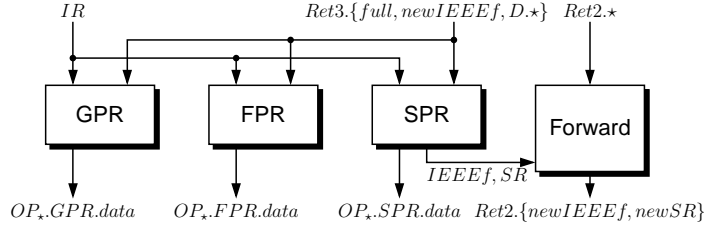


Figure 4.33: Register files

4.7.1 Forwarding

No forwarding is done from the write accesses in the retire sub-phase $Ret3$ writing the instruction results to the read accesses in the decode sub-phase $D1$ reading the operands. This is not needed for the correctness as stated by the following theorem.

Theorem 4.9. *The write accesses to the register files in the retire sub-phase $Ret3$ need not to be forwarded to the read accesses to the register files ports in the decode sub-phase $D1$.*

Proof. Let I_D be an active instruction in the decode phase that depends on the result of an instruction I_R . If I_R has written its result to the register file before I_D starts its read access to the register file, no forwarding has to be done because the register file RAM will return the result of the instruction I_R . Thus, assume that I_R is still active (i.e., has not yet written the register file) at the time I_D starts its read access to the register file. Then the producer table entry of the operand of I_D that depends on I_R still contains the tag of I_R . Hence, the instruction I_D will check the ROB if it contains the result of I_R .

If I_R has not completed before the instruction I_D starts the ROB read access in the operand-read-context, it follows from the construction of the forwarding circuit for the ROB (see section 4.6.3) that the result of I_R is forwarded to I_D . Thus, assume that I_R has completed before I_D starts the read access to the ROB. Thus, the ROB contains the result of I_R . It remains to show that the read access to the ROB in the operand-read-context of the instruction I_D returns the result of I_R .

At the time I_D is in the first stage of decode sub-phase $D1$, the instruction I_R has not yet written the register file, i.e., I_R is active. Thus, from lemma 4.6 it follows that no instruction older or equal to I_D and younger than I_R can have the same tag as I_R and therefore will not overwrite to the ROB entry of I_R and hence the ROB entry of I_R is still valid at the time I_D starts the ROB read access in the operand-read-context.

If issue is done in one cycle, only the write access to the ROB in the allocation-context of the instruction I_D can overtake the read access to the ROB in the operand-read-context of I_D as discussed in section 4.6.3. Since I_D has not the same tag as I_R the write access in the allocation-context will not overwrite the entry of I_R . Hence, the read access returns the value of the ROB at the time the read access has been started, i.e., the valid result of I_R .

Let c_{D2} be the number of cycles needed for the read access to the ROB in the operand-read-context. If issuing is done in multiple cycles also the write accesses to the ROB in the allocation-context of the $c_{D2} - 1$ instructions following I_D can overtake the read access to the ROB of I_D . From lemma 4.6 it follows that these instructions

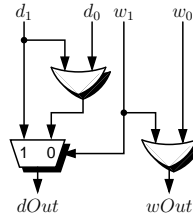


Figure 4.34: Forwarding circuit IEEEfC

cannot have the same tag as I_R and therefore I_D reads the valid result of I_R . \square

IEEEf

Let I be an instruction in the retire sub-phase *Ret2*. As discussed in section 4.5 the special purpose register must return the values of the special registers *IEEEf* and *SR* at the time the instruction I enters the retire sub-phase *Ret3*. The register files are only written in the retire sub-phase *Ret3*. Thus, to obtain the value of the registers at the time the instruction I enters the sub-phase *Ret3* it suffices to read the register when I enters the sub-phase *Ret2* and forward the writes to these registers of all older instructions in the sub-phase *Ret2*.

The forwarding for the register *SR* can be done using a standard forwarding tree (see section 2.6.1). This forwarding tree combines the updates to the register *SR* of the instructions in *Ret2* which are older than I in parallel to I 's read access to the register *SR*. The outputs of the forwarding tree and the register access are then combined to obtain the bus *newSR* containing the value of the register *SR* at the time the instruction I will enter the sub-phase *Ret3*.

To forward the register *IEEEf* a modified forwarding tree must be used as the register *IEEEf* can be written in two different ways: if the instruction which retires is a floating point operation, the new value of *IEEEf* is computed by OR-ing the floating point exception bits of the instruction to the old value of *IEEEf* [IEEE]; if the instruction explicitly writes the register *IEEEf* (e.g., a moveI2S instruction), the old value is overwritten.

The circuit IEEEfC (see figure 4.34) combines the updates to the register *IEEEf* of two succeeding instruction. For this it uses the types w and data d for updating the register *IEEEf* of an instruction. The type w of an instruction is one if the instruction explicitly writes the register *IEEEf* with the data d ; if w is zero, the data d is to be OR-ed to the old value.

Let w_1 and d_1 be type and data of an instruction I_1 that succeeds an instruction I_0 with type w_0 and data d_0 . The combined type $wOut$ of the instructions I_1 and I_0 is the OR the types indicating that any of the instructions I_1 and I_0 will directly write the register. If w_1 is one, the instruction I_1 will overwrite the access done by I_0 . Thus, in this case the combined data $dOut$ is the data d_1 of instruction I_1 . If w_1 is zero the data d_1 will be OR-ed to the data d_0 of the instruction I_0 . In this case, if w_0 is active, the preceding instructions will be overwritten with the OR of both instructions, otherwise the OR of both instructions will be OR-ed to the old value. Thus, updating the register *IEEEf* with the combined values $wOut$ and $dOut$ has the same effect as the sequential updates of instructions I_0 and I_1 .

Using the fact that the circuit **IEEEfC** computes an associative function (proven in the following lemma 4.10) a forwarding tree can be built with the circuit **IEEEfC** at the nodes. Using all instructions in the retire sub-phase *Ret2* as inputs, this tree computes the combined update of these instructions to the register *IEEEf*. The output of the special purpose register RAM can then be combined with the output of the forwarding tree by another **IEEEfC** circuit obtaining the value *newIEEEf* of the register at the time the youngest instruction in the tree retires.

The type *w* and data *d* for every instruction can be computed as follows. A stage in the sub-phase *Ret2* contains a *moveI2S* instruction which writes the register *IEEEf* and therefore will explicitly write the register, if the stage is full and the signal *ROB.writeIEEEf* is active. Thus, the type *w* can be computed as:

$$w := full \wedge ROB.writeIEEEf.$$

If *w* is one, the data that has to be written into the register *IEEEf* is located in the lowest 5 bits of the result bus *ROB.data*. Otherwise the floating point exception bits read from the ROB *ROB.IEEEf* are used. If the stage is not full the floating point exception bits of the stage are set to zero. Then the data *d* used to update the register *IEEEf* are:

$$d := \begin{cases} ROB.D.data.lo[4 : 0] & \text{if } w \\ ROB.IEEEf \wedge full & \text{if } \bar{w} \end{cases}$$

Hence, if the stage is not full the register *IEEEf* is not changed as it is OR-ed with a constant zero. Especially if no instruction is retiring the bus *newIEEEf* contains the value stored in the register file.

Lemma 4.10. *The circuit **IEEEfC** computes an associative function.*

Proof. Without loss of generality let *d* be only one bit wide. Let (w_0, d_0) , (w_1, d_1) , and (w_2, d_2) be in $\{0, 1\}^2$. The function \circ computed by the circuit **IEEEfC** is:

$$\begin{aligned} (w_1, d_1) \circ (w_0, d_0) &= (w_1 \vee w_0, w_1 ? d_1 : (d_1 \vee d_0)) \\ &= (w_1 \vee w_0, w_1 d_1 \vee \bar{w}_1 (d_1 \vee d_0)) \end{aligned}$$

It holds:

$$\begin{aligned} &((w_2, d_2) \circ (w_1, d_1)) \circ (w_0, d_0) \\ &= (w_2 \vee w_1, w_2 d_2 \vee \bar{w}_2 (d_2 \vee d_1)) \circ (w_0, d_0) \\ &= (w_2 \vee w_1 \vee w_0, (w_2 \vee w_1)(w_2 d_2 \vee \bar{w}_2 (d_2 \vee d_1)) \\ &\quad \vee \overline{(w_2 \vee w_1)}(w_2 d_2 \vee \bar{w}_2 (d_2 \vee d_1) \vee d_0)) \\ &= (w_2 \vee w_1 \vee w_0, w_2 d_2 \vee w_2 w_1 d_2 \vee w_2 \bar{w}_2 (d_2 \vee d_1) \vee \bar{w}_2 w_1 (d_2 \vee d_1) \\ &\quad \vee \bar{w}_2 \bar{w}_1 (w_2 d_2 \vee \bar{w}_2 (d_2 \vee d_1) \vee \bar{w}_2 d_0 \vee w_2 d_0)) \\ &= (w_2 \vee w_1 \vee w_0, w_2 d_2 \vee \bar{w}_2 w_1 (d_2 \vee d_1) \vee \bar{w}_2 \bar{w}_1 (d_2 \vee d_1 \vee d_0)) \\ &= (w_2 \vee w_1 \vee w_0, w_2 d_2 \vee \bar{w}_2 (w_1 d_2 \vee \bar{w}_1 d_2 \vee w_1 d_1 \vee \bar{w}_1 (d_1 \vee d_0))) \\ &= (w_2 \vee w_1 \vee w_0, w_2 d_2 \vee \bar{w}_2 (d_2 \vee w_1 d_1 \vee \bar{w}_1 (d_1 \vee d_0))) \\ &= (w_2, d_2) \circ (w_1 \vee w_0, w_1 d_1 \vee \bar{w}_1 (d_1 \vee d_0)) \\ &= (w_2, d_2) \circ ((w_1, d_1) \circ (w_0, d_0)) \end{aligned}$$

□

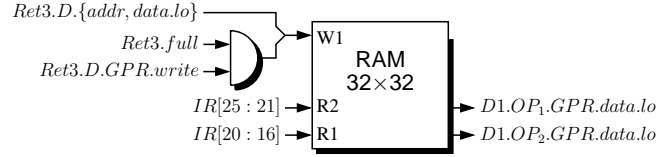


Figure 4.35: General purpose register file

Cost and Delay

The cost and delay of the circuit **IEEEfC** are:

$$\begin{aligned} D(\text{IEEEfC}) &\leq D_{OR} + D_{MUX}, \\ C(\text{IEEEfC}) &\leq 5 \cdot C_{MUX} + 6 \cdot D_{OR}. \end{aligned}$$

Since the delay of the forwarding tree used to compute the signals $newIEEEf$ and $newSR$ are only logarithmic in the number of stages of $Ret2$, the circuit is not assumed to be critical compared to the register file access. The delay of the output $SPR.newIEEEf$ and $SPR.newSR$ then are:

$$\begin{aligned} D(\text{SPR.newIEEEf}) &\leq D(\text{SPR-RF}) + D(\text{IEEEfC}), \\ D(\text{SPR.newSR}) &\leq D(\text{SPR-RF}) + D_{MUX}. \end{aligned}$$

Let c_{RF} be the number of cycles needed for an register file read access and let c_{Ret2} be the number of cycles needed for the retire sub-phase $Ret2$. Then the cost of the forwarding for the register files are:

$$\begin{aligned} C(\text{RF-Forward}) &\leq C(\text{Forward-Tree}(5, 32, c_{Ret2}, c_{RF})) \\ &\quad + c_{Ret2} \cdot (5 \cdot C_{MUX} + 6 \cdot C_{AND} + C(\text{IEEEfC})) \\ &\quad + c_{RF} \cdot \lceil (c_{Ret2} \cdot 12 + 6) / 2 \rceil \cdot C_{REG}. \end{aligned}$$

4.7.2 General Purpose Register File

The GPR consists of 32 register each of which is 32 bits wide. The address computation for the read accesses of the GPR is rather simple. For all instructions which access the GPR, it holds (see appendix A):

$$\begin{aligned} OP_1.GPR.addr[4 : 0] &:= IR[25 : 21], \\ OP_2.GPR.addr[4 : 0] &:= IR[20 : 16]. \end{aligned}$$

The design of the GPR is straightforward (see figure 4.35). It is a RAM block with 32 entries each consisting of 32 bits and two read and one write port. The write signal may only be active if an instruction is in the retire sub-phase $Ret3$. Only the low part of the data bus is used for read and write accesses. No forwarding is necessary for the GPR as stated in theorem 4.9.

The write path of does not influence any critical path or the number of pipeline stages. Therefore, only the delay of the read access of the GPR is taken into account. The same holds true for all other register files. Hence, the delay of the GPR is estimated as:

$$D(\text{GRP-RF}) \leq D(\text{RAM}(32, 32, 2, 1)).$$

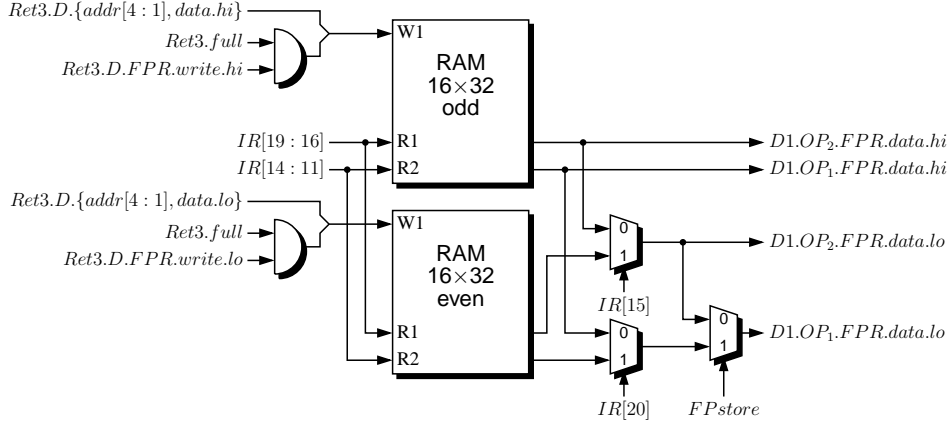


Figure 4.36: Floating point register file

Let c_{RF} be the delay of a register file access. The cost of the GPR environment is:

$$C(\text{GRP-RF}) \leq C(\text{RAM}(32, 32, 2, 1, c_{RF})) + C_{AND}.$$

4.7.3 Floating Point Register File

The FPR holds 32 single precision registers with 32 bits. The 16 pairs of even and odd registers (i.e., the pairs 0 and 1 to 30 and 31) can also be accessed as 64 bits wide double precision registers using the address of the even register. Double precision accesses with an odd address raise an illegal instruction interrupt in the decode phase.

To support the two access modes, the FPR is divided in two register files of 16 entries for even and odd registers (see figure 4.36). The lowest bit of the addresses of the two read ports select the outputs of the RAMs for the low parts of the two outputs. The high part always uses the odd RAM, as 64 bit register accesses always have even addresses.

The FPR uses the two separate write signals. The signal $D.FPR.write.lo$ controls the RAM block containing the odd registers, $D.FPR.write.hi$ controls the RAM block containing the even registers. For 64 bit results (indicated by $D.dbl$), the write signals for both RAM blocks must be active. For 32 bit results only the write signal for the addressed RAM block may be active. Analogously to the GPR the write port is only active if the retire sub-phase $Ret3$ is full. The write signals $D.FPR.write.lo$ and $D.FPR.write.hi$ are computed during the sub-phase $Ret2$ as:

$$\begin{aligned} D.FPR.write.lo &:= D.FPR.write \wedge \overline{D.addr[0]}, \\ D.FPR.write.hi &:= D.FPR.write \wedge (D.addr[0] \vee D.dbl). \end{aligned}$$

The low part of the result $D.data.lo$ is connected to the RAM block for the even registers, the high part $D.data.hi$ is connected to the RAM block for the odd registers. Note that in order to write 32 bit results into the results must be available on both the high and the low part of the data bus (see table 4.3 on page 63).

For all instructions which read the FPR except floating point stores the address of the first operand is $IR[20 : 16]$, the address of the second operand is $IR[15 : 11]$ (see appendix A). The floating point store instruction (indicated by the signal $FPstore$

computed in the **Decode** circuit) uses the the bits $IR[20 : 16]$ as second operand. Thus, the address of the floating point operands is:

$$\begin{aligned} OP_1.FPR.addr &:= IR[20 : 16], \\ OP_2.FPR.addr &:= \begin{cases} IR[20 : 16] & \text{if } FPstore \\ IR[15 : 11] & \text{if } \overline{FPstore} \end{cases}. \end{aligned}$$

To hide the delay of the signal $FPstore$, the FPR is accessed under the assumption that the instruction is not a floating point store. If the instruction signal $FPstore$ is active, the output for the first operand is used as output for the second operand. This is done by the multiplexer controlled by the signal $FPstore$ in figure 4.36.

The selection of the operands using the signal $FPstore$ can be combined with the select circuit in the circuit **OpGen** (see section 4.1.4), which selects between the outputs of the different register files. Hence, instead of a 4 input select circuit an 5 input select circuit is used. The difference in delay and cost are added to the floating point register file. It holds:

$$\begin{aligned} D(\text{Sel}(5)) - D(\text{Sel}(4)) &= D_{OR}, \\ C(\text{Sel}(5)) - C(\text{Sel}(4)) &= D_{AND} + D_{OR}. \end{aligned}$$

Hence, the delay of the last multiplexer for the computation of $D1.OP_1.FPR.data.lo$ can be replaced by D_{OR} . The overall delay of the FPR RAM is:

$$D(\text{FPR-RF}) \leq D(\text{RAM}(16, 32, 2, 1)) + D_{MUX} + D_{OR}.$$

Let c_{RF} be the number of cycles for a register file access. The cost of the FPR including the computation of the write signals is approximated by:

$$\begin{aligned} C(\text{FPR-RF}) &\leq 2 \cdot C(\text{RAM}(16, 32, 2, 1, c_{RF})) + 4 \cdot C_{AND} + C_{OR} \\ &\quad + 64 \cdot C_{MUX} + 32 \cdot (C_{AND} + C_{OR}). \end{aligned}$$

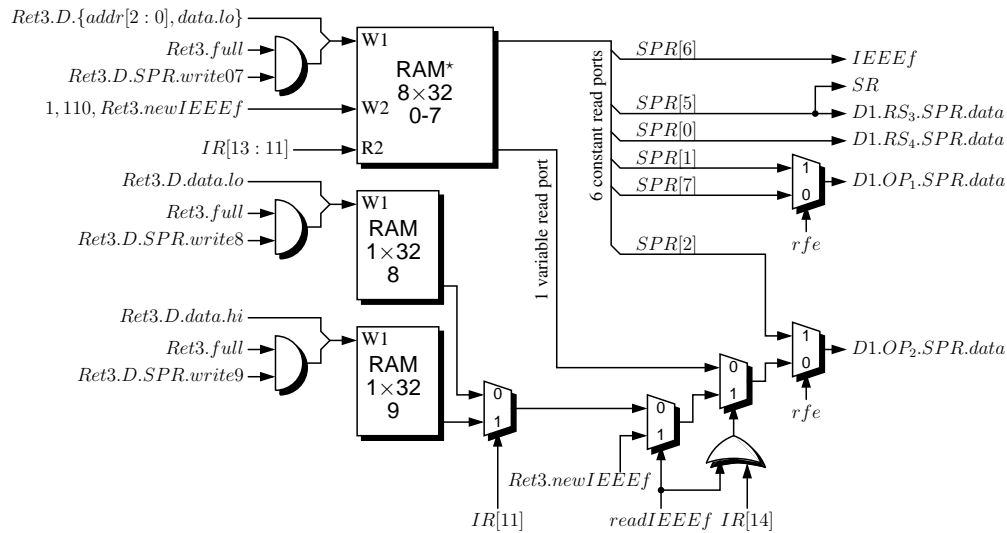
4.7.4 Special Purpose Register File

The special purpose register file consists of 10 register which are summarized in table 4.7. The register 0 to 4 are used for interrupt handling. The rounding mode for floating point operations is stored in register 5. Register 6 collects the floating point exception flags. The result of floating point compares are stored in register 7, which may then be used by floating point branches. The registers 8 and 9 are used to store the 64 bit wide results of integer multiplication and divisions.

When an interrupt occurs, the special purpose register file must perform the following actions:

$$\begin{aligned} SR &:= 0, \\ ESR &:= SR, \\ EPC &:= Ret3.ePC, \\ ECA &:= Ret3.MCA, \\ EData &:= Ret3.eData. \end{aligned}$$

| addr | name | purpose |
|------|-------|----------------------------------|
| 0 | SR | status register (interrupt mask) |
| 1 | ESR | exception status register |
| 2 | EPC | exception program counter |
| 3 | ECA | exception cause register |
| 4 | EData | exception data register |
| 5 | RM | floating point rounding mode |
| 6 | IEEEf | IEEE interrupt flags |
| 7 | FCC | floating point condition code |
| 8 | LO | LO register |
| 9 | HI | HI register |

Table 4.7: Special purpose registers**Figure 4.37:** Special purpose register file

The implementation of this operation is not discussed in detail here. It is assumed that a special RAM block can be used which performs these operations if the clear signal is activated. A construction for such a RAM block using discrete gates can be found, e.g., in [Krö99].

The circuit SPR is depicted in figure 4.37. It consists of one RAM block with 8 entries and two RAM blocks with one entry⁷ for the registers *LO* and *HI*. Similar to an even / odd pair of floating point registers these two registers can be written in parallel to store the 64 bit wide result of an integer multiplication or division.

The following four instruction types have SPR registers as operands (see appendix A):

- moveS2I instructions: These instructions read an arbitrary special register as second operand and save it in an general purpose register. The address of the

⁷A RAM block with only one entry is basically a single register and therefore need no address entry. For the sake of description they are yet treated as RAM blocks.

special register is encoded in the bits [15 : 11] of the instruction word.

- return-from-exception (rfe) instructions: The rfe instruction sets the PC to the value of the special register *EPC* and copies the special register *ESR* into the special register *SR*. It uses the register *ESR* as first and *EPC* as second operand.
- branch on floating point condition code (BC1) instructions: The *BC1* instructions are conditional branches that depend on the value of the special register *FCC*. This register is used as first operand of the instruction.
- floating point instructions: These instructions depend on the special registers *RM* and *SR* as third and fourth operand.

Hence, the addresses of the four special purpose register operands are:

$$\begin{aligned}
 OP_1.SPR.addr &:= \begin{cases} 00001 & \text{if } rfe \\ 00111 & \text{if } \overline{rfe} \end{cases}, \\
 OP_2.SPR.addr &:= \begin{cases} 00010 & \text{if } rfe \\ IR[15 : 11] & \text{if } \overline{rfe} \end{cases}, \\
 OP_3.SPR.addr &:= 00101, \\
 OP_4.SPR.addr &:= 00000.
 \end{aligned}$$

To implement this, the SPR has one variable read port and five constant read ports for the entries *EPC*, *ESR*, *FCC*, *RM*, and *SR*. Similar to the floating point store for the FPR, a multiplexer controlled by the signal *rfe* indicating an rfe-instruction selects the data output for the first and second operand.

If the instruction is a *moveS2I* instruction which reads the register *IEEEf* (indicated by the signal *readIEEEf*), the SPR returns the current content of the bus *Ret3.newIEEEf* and not the content of the register. Recall that an instruction reading the register *IEEEf* waits in the pipeline stage of the decode phase in which the SPR return the result. These instructions wait there until all preceding instructions have retired (see section 4.1.7). As soon as this happens, the SPR must return the correct value of the register *IEEEf* without restarting the read access. This can be done by returning the bus *Ret3.newIEEEf* which then contains the correct value of the register (see section 4.7.1).

Every instruction *I* which enters the retire sub-phase *Ret2* must read the registers *SR* and *IEEEf* from the SPR in order to compute the busses *newSR* and *newIEEEf* containing the value of these registers at the time the instruction *I* enters the sub-phase *Ret3* (see section 4.7.1). The SPR has already a constant read port for the register *SR*, thus only one additional constant read port is needed for reading the register *IEEEf*.

The registers 8 and 9 must be written at the same time to store the 64 bit result of integer multiplications or divisions. Therefore, these registers are treated separately from the registers 0 to 7. The registers 8 and 9 are divided into two RAM blocks, which may be written in parallel. If the register 9 is written with a 32 bit result (by the

instruction *moveI2S*), the data must be on the high part of the input data bus. Since *moveI2S* instructions use the ALU which writes its results to both the high and low parts of the result bus, this is already guaranteed (see table 4.3 on page 63).

To compute the write signals for the two RAM blocks the signal $D.dbl$ computed by the circuit **DestCmp** is used. If the result is stored in the SPR this signal indicates an integer multiplication or division. Thus, if $D.dbl$ is active the address bits of the destination address can be ignored. If $D.dbl$ is not active, the bits 0 and 3 of the address are used to decide which RAM block is written. As for the FPR, the write signals are assumed to be computed during the retire sub-phase *Ret2*.

$$\begin{aligned} SPR.write07 &:= D.SPR.write \wedge \overline{D.dbl} \wedge \overline{D.addr[3]}, \\ SPR.write8 &:= D.SPR.write \wedge (D.dbl \vee (D.addr[3] \wedge \overline{D.addr[0]})), \\ SPR.write9 &:= D.SPR.write \wedge (D.dbl \vee (D.addr[3] \wedge (D.addr[0]))). \end{aligned}$$

The *IEEEf* register is written using an additional write port with constant address. If the retire sub-phase is not full, the value of the input bus *Ret3.newIEEEf* is the unchanged value of the previous instruction (see section 4.7.1). Thus, this write port can be active in every cycle. Due to the construction of the forwarding circuit, the inputs of both write ports are identical if an instructions writes the register *IEEEf* explicitly. Hence, the priority of the write ports is irrelevant.

The rightmost multiplexers in figure 4.37 can be incorporated into select circuit of the circuit **OpGen** analogously to the FPR. Hence, the delay and cost of this multiplexers can be replaced by D_{OR} respectively $C_{AND} + C_{OR}$. The delay of the SPR RAM is:

$$\begin{aligned} D(\text{SPR-RF}) &\leq \max\{D(\text{RAM}(8, 32, 2, 2)), D(\text{RAM}(1, 32, 1, 1)) + 2 \cdot D_{MUX}\} \\ &\quad + D_{MUX} + D_{OR}, \end{aligned}$$

The cost and delay of the RAM block for the SPR registers 0 to 7 are approximated by the cost and delay of RAM block with 2 variable read and write ports. Let c_{RF} be the number of cycles needed for a register files access. Then the cost and delay of the circuit **SPR** can be approximated by:

$$\begin{aligned} C(\text{SPR-RF}) &\leq C(\text{RAM}(8, 32, 2, 2, c_{RF})) + 2 \cdot C(\text{RAM}(1, 32, 1, 1, c_{RF})) \\ &\quad + 5 \cdot C_{AND} + C_{OR} + 3 \cdot 32 \cdot C_{MUX} + 2 \cdot 32 \cdot (C_{AND} + C_{OR}). \end{aligned}$$

4.7.5 Cost and Delay

The read accesses to the SPR in the retire sub-phase *Ret2* are not assumed to be critical for the overall delay of the register file environment, as they use read ports with constant addresses. For the register file accesses in the decode sub-phase *D1* no forwarding is necessary (see theorem 4.9). Hence, the delay of the register file environment and the number of cycles needed for a register file access c_{RF} is:

$$\begin{aligned} D(\text{RF}) &\leq \max\{D(\text{GPR-RF}), D(\text{FPR-RF}), D(\text{SPR-RF})\}, \\ c_{RF} &= \lceil D(\text{RF})/\delta \rceil. \end{aligned}$$

The cost of the register files is:

$$C(\text{RF}) \leq C(\text{GPR-RF}) + C(\text{FPR-RF}) + C(\text{SPR-RF}).$$

4.8 Producer Table Environment

The producer tables are similar to their corresponding register files, but have one additional read and one additional write port. The new write port is used during the decode sub-phase *D1* to write the tag of the instruction into the producer table. The additional read port is used during the retire sub-phase *Ret2* to check if no succeeding instruction has overwritten the entry. All producer tables have a reset signal which sets all valid bits to one.

The environment of the RAM blocks itself is basically the same as for the register files and therefore not discussed in detail. The address and write signals for the additional ports are computed in advance by the circuit **DestCmp**. The priority of the new write access during *D1* is higher than the priority of the write access during *Ret3*.

The entries of the producer tables consist of the valid bit and the tag for the corresponding register file entries. The entries for the odd registers of the floating point producer table and the register 9 of the special purpose producer table need an extra bit *dbl* that indicates whether this register is written by a double precision result. The output $OP_i.\mathfrak{R}.dbl$ for $i \in \{1, 2\}$ and $\mathfrak{R} \in \{FPR, SPR\}$ returns the value of this bit, if one of these registers is addressed by the operand, otherwise 0. The output is needed by the reservation stations to decide if it needs to use the high or the low part to update an operand. The value of the bit *dbl* is set to the value of the signal $D.\mathfrak{R}.dbl$ computed by the circuit **DestCmp** if one of the registers is written during the decode sub-phase *D1*. It is set to 0 if the producer table is written during the retire sub-phase *Ret3*.

4.8.1 Forwarding

The producer tables are accessed in four different contexts (operand-read, updating, checking, and retiring). In the decode sub-phase *D1* the producer tables are read to obtain the valid bit and the tag of the operands (operand-read-context). Also in the decode sub-phase *D1* the tag of the new instruction is written into the producer table entry of the destination register of the instruction (updating-context). In the retire sub-phase *Ret2* the producer table entry of the destination register is read again to check whether a succeeding instruction will write the same register (checking-context). If no such instruction exists, in the retire sub-phase *Ret3* the producer producer table entry of the destination register is set valid in order to flag valid register file content (retiring-context). The following lemmas summarize the dependencies between the accesses.

Lemma 4.11. *The result of the read access to the producer table in the checking-context for an instruction I must take all write accesses to the producer table in the updating-context into account that are started before the instruction I writes the producer table in the retiring-context.*

Proof. All write accesses to the producer table that are started before the read access in the checking-context starts are taken into account by construction of the RAM. Hence,

let an instruction I_1 write the producer table in the updating-context after an older instruction I_0 has started the read access in the checking-context and before I_0 starts the write access in the retiring-context. Let I_1 be the first instruction succeeding I_0 that writes the same destination register as I_0 . If the write in the updating-context by I_1 is not forwarded to the read access in the checking-context by I_0 , the instruction I_0 will read its own tag. Thus, I_0 will set the valid bit in the retiring-context, even if the content of the register is not valid as it will eventually be overwritten by I_1 . \square

Theorem 4.12. *The write access to the producer table in the retiring-context does not have to be forwarded to the read access in the checking-context.*

Proof. Let I_0 be an instruction which writes its result into the register file in the retiring-context and I_1 be an instruction which simultaneously reads the producer table in the checking-context. Assume the instructions I_0 and I_1 write the same register. Then both instructions have written their tags to the producer table entry of this register in the updating-context. As decode and retire are done in order, the instruction I_1 has written the producer table entry after the instruction I_0 in the updating-context. This update of the instruction I_1 must have been forwarded to the read access in the checking-context of instruction I_1 (see lemma 4.11). Thus, the instruction I_0 cannot have read its own tag in the check-context. Thus, the instruction I_0 will not write the producer table in the retire context. \square

Let c_{TC} denote the number of cycles it takes from the start of the forwarding of a write access in the updating-context to the end of the retire sub-phase *Ret2*. Then in order to compute the content of the producer table at the time an instruction enters the retire sub-phase *Ret2*, the write access in the updating-context must be delayed by $c_{TC} - 1$ cycles (see section 2.6.3). As the content of the register does not depend on the write access in the retire context (see theorem 4.12), this write port does not need to be delayed.

Theorem 4.13. *The read access in the operand-read-context does not depend on the write access in the retiring-context.*

The accesses in the operand-read-context and the retiring-context correspond to the read and write accesses to the register files. Instead of forwarding the write access the data are read out of the ROB. Thus, this theorem can be proven analogously to the theorem 4.9 on page 95.

Lemma 4.14. *The read access in the operand-read-context must take exactly the writes in the updating-context into account that are started by preceding instruction.*

Proof. An instruction must not depend on a succeeding instruction; therefore the write access in the updating-context by succeeding instruction must not be forwarded. Since an instruction may depend on any preceding instruction the writes in the updating-context of all preceding instructions must be forwarded. \square

Figure 4.38 details the forwarding for the producer tables. The write access in the retiring-context does not interfere with any other accesses and is therefore directly connected to the RAM blocks of the producer tables.

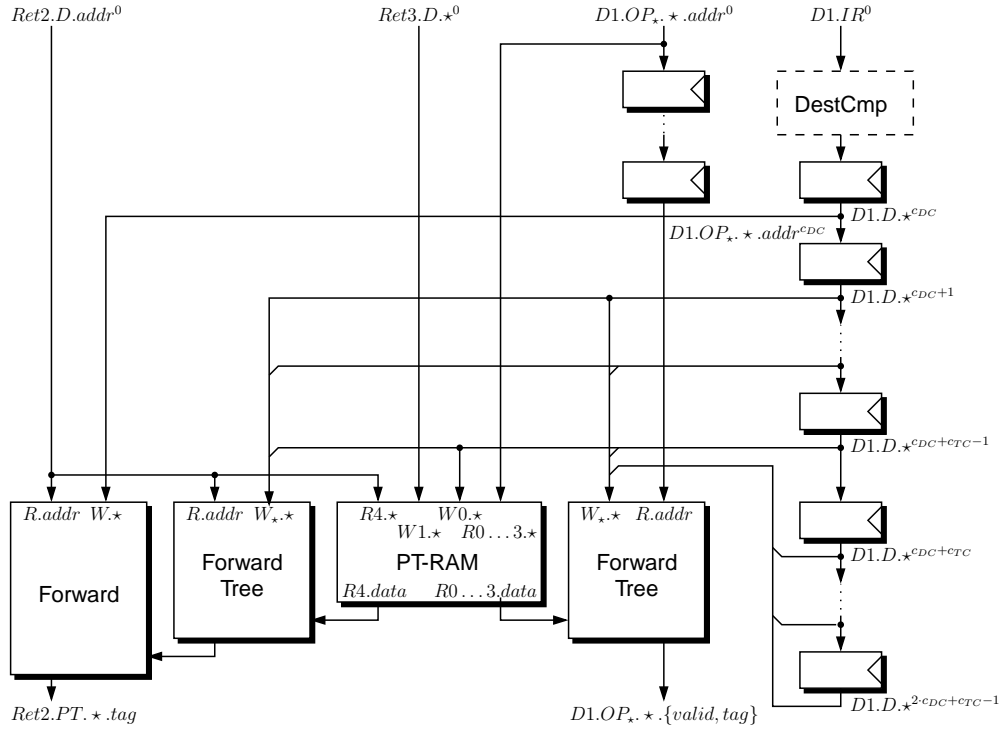


Figure 4.38: Forwarding of the producer table

Due to the circuit **DestCmp** the address, data, and write signal $D1.D.*$ for the write access in the updating-context are not known before cycle c_{DC} of $D1$. At this cycle, the write access in the updating-context is forwarded to the read access in the checking-context (to address $Ret2.D.addr$) analogous to figure 2.13 on page 24 using the forwarding circuit and the forwarding tree on the left side of figure 4.38. Note that in order to realize this forwarding, the write access has to be delayed by another $c_{TC} - 1$ cycles and hence enters the RAM blocks in cycle $c_{DC} + c_{TC} - 1$.

The read access in the operand-read-context enters the RAM blocks in cycle 0 of the decode sub-phase $D1$. It must take all write accesses in the updating-context of preceding instructions into account. Since the write access in the updating-context is not started before cycle $c_{DC} + c_{TC} - 1$, the RAM blocks do not return the data of the write accesses in the updating-context of the $c_{DC} + c_{TC} - 1$ preceding instructions. These write accesses must be forwarded using a forwarding tree.

The address and write signal of the write access in the updating-context is not known before cycle c_{DC} of $D1$. Therefore the forwarding tree is delayed by c_{DC} cycles and compares the address of the read access in the operand-read-context in cycle c_{DC} $D1.OP_*.*.addr^{c_{DC}}$ with the addresses of the write access in the updating-context of the instructions in the $c_{DC} + c_{TC} - 1$ succeeding stages (i.e., stages $c_{DC} + 1$ to $2 \cdot c_{DC} + c_{TC} - 1$). Note that the delay of the circuit **DestCmp** is relatively small in comparison to the delay of the RAM access. Therefore, it can be assumed that the forwarding tree needs less cycles than the RAM access even if it is delayed by c_{DC} cycles.

4.8.2 Cost and Delay

Let $b := l_{ROB} + 1$ be the width of the producer table entries. The delay of the producer tables without forwarding can be estimated as:

$$\begin{aligned}
 D(\text{GRP-PT}) &\leq D(\text{RAM}(32, b, 3, 2)), \\
 D(\text{FPR-PT}) &\leq D(\text{RAM}(16, b + 1, 3, 2)) + D_{MUX} + D_{AND}, \\
 D(\text{SPR-PT}) &\leq \max\{D(\text{RAM}(8, b, 3, 3)), D(\text{RAM}(1, b + 1, 2, 2)) + 2 \cdot D_{MUX}\} \\
 &\quad + D_{MUX} + D_{AND}, \\
 D(\text{PT-RAM}) &\leq \max\{D(\text{GPR-PT}), D(\text{FPR-PT}), D(\text{SPR-PT})\}.
 \end{aligned}$$

The delay of the read access in the operand-read context is:

$$D(D1.OP_{\star} \cdot \{valid, tag\}) \leq D(\text{PT-RAM}) + D_{MUX}.$$

Let c_{TC} be the number of cycles needed from forwarding the write port in the updating-context to the end of the retire sub-phase *Ret2*. The producer table access is assumed to be the critical path of the retire sub-phase *Ret2*, otherwise it can be delayed such that it has the same delay as the critical path. In any case c_{TC} is determined by the delay of the forwarding circuit for 5 address bits and the delay of the circuit *TagCheck*:

$$c_{TC} = \lceil (D(\text{Forward}(5)) + D(\text{TagCheck})) / \delta \rceil.$$

If $c_{TC} = 1$, the write in the updating-context does not have to be delayed and no additional forwarding tree is needed for the read access in the checking-context. In this case the forwarding increases the delay of the read access in the checking-context by the mux inside the forwarding circuit needed for merging the outputs of forwarding circuit and RAM, otherwise by the two muxes for first merging the output of the RAM with the output of the forwarding tree inside the forwarding tree and for merging with the output of the forwarding circuit (see figure 4.38). Thus, the delay of the producer table access in the checking-context is:

$$D(\text{Ret2.PT} \cdot \{tag\}) \leq D(\text{PT-RAM}) + \begin{cases} D_{MUX} & \text{if } c_{TC} = 1 \\ 2 \cdot D_{MUX} & \text{if } c_{TC} > 1 \end{cases}.$$

The maximum delay of the read accesses to the producer table and the number of cycles c_{PT} needed is:

$$\begin{aligned}
 D(\text{PT}) &\leq \max\{D(D1.OP_{\star} \cdot \{valid, tag\}), D(\text{Ret2.PT} \cdot \{tag\})\}, \\
 c_{PT} &= \lceil D(\text{PT}) / \delta \rceil.
 \end{aligned}$$

The costs of the producer tables without forwarding are:

$$\begin{aligned}
C(\text{GRP-PT}) &\leq C(\text{RAM}(32, b, 3, 2, c_{PT})) + C_{AND}, \\
C(\text{FPR-PT}) &\leq C(\text{RAM}(16, b, 3, 2, c_{PT})) + C(\text{RAM}(16, b+1, 3, 2, c_{PT})) \\
&\quad + 6 \cdot C_{AND} + C_{OR} + 6 \cdot C_{MUX} \\
&\quad + 2 \cdot b \cdot C_{MUX} + 2 \cdot 2 \cdot b \cdot C_{AND}, \\
C(\text{SPR-PT}) &\leq C(\text{RAM}(8, b, 3, 3, c_{PT})) \\
&\quad + C(\text{RAM}(1, b, 2, 2, c_{PT})) + C(\text{RAM}(1, b+1, 2, 2, c_{PT})) \\
&\quad + 12 \cdot C_{AND} + 2 \cdot C_{OR} + 3 \cdot b \cdot C_{MUX} + 4 \cdot b \cdot C_{AND}, \\
C(\text{PT-RAM}) &\leq C(\text{GPR-PT}) + C(\text{FPR-PT}) + C(\text{SPR-PT}).
\end{aligned}$$

The number of cycles needed for the computation of the destination registers c_{DC} is:

$$c_{DC} = \lceil D(\text{DestComp})/\delta \rceil.$$

The forward circuit for the read access in the checking-context consists of a forward circuit and a forwarding tree with c_{TC} inputs for each PT. The forward circuit for the read access in the operand-read-context consists of a forwarding tree with $c_{TC} + c_{DC} - 1$ leaves for each operand output of the PTs (note that the high parts of the FPR and the SPR uses only 4 address bits). The total cost for the forwarding circuit of the PT are:

$$\begin{aligned}
C(\text{PT-Forward}) &\leq 3 \cdot C(\text{Forward}(5, b, c_{PT}, c_F(5))) \\
&\quad + 3 \cdot C(\text{Forward-Tree}(5, b, c_{TC}, c_{PT})) \\
&\quad + 4 \cdot C(\text{Forward-Tree}(5, b, c_{DC} + c_{TC} - 1, c_{PT})) \\
&\quad + 6 \cdot C(\text{Forward-Tree}(4, b, c_{DC} + c_{TC} - 1, c_{PT})) \\
&\quad + (c_{DT} + c_{TC}) \cdot 38 \cdot C_{REG}.
\end{aligned}$$

The total cost for the producer tables is:

$$C(\text{PT}) \leq C(\text{PT-RAM}) + C(\text{PT-Forward}).$$

Chapter 5

Memory Unit

This section describes the memory unit of the $DLX_{\pi+}$. An overview of the memory unit is given in section 5.1. Sections 5.2 to 5.8 describe the non-blocking data cache used in the memory unit.

5.1 Overview

The memory unit handles all load and store accesses to the main memory. The unit presented in this thesis does not support virtual memory, i.e. the addresses sent by the processor can be directly used to address the main memory. Page fault interrupts are not computed by the memory unit but are assumed to be computed by the main memory. Furthermore the memory unit is not assumed to be able to write to the instruction memory. Thus, no cache coherency protocol is needed between the instruction and the data cache. Lines in the cache do not have to be invalidated.

The memory unit is divided into the three circuits **Sh4S**, **DCache**, and **Sh4L** (see figure 5.1). The data cache which does the actual memory access is contained in the circuit **DCache**. All accesses to the data cache must be aligned to word-addresses. The adaption for instructions which do not access whole words are done by the circuits **Sh4S** and **Sh4L**. In contrary to the memory unit of the DLX by Müller and Paul [MP00] double word accesses are not supported by the MIPS R3000 ISA.

The memory unit gets as input the control bus *con* that defines the type of the access (including the signal *write* indicating a store and an immediate constant *imm*), the tag of the instruction, and the two operands OP_1 and OP_2 . The circuit **Sh4S** first

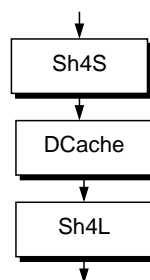


Figure 5.1: Memory Unit

computes the effective address $addr$ of the access as sum of the first operand and the immediate constant. If the memory access is misaligned, the circuit **Sh4S** raises the data access misaligned interrupt $Dmal$. In this case the instruction is sent directly to the circuit **Sh4L**. Thus, the data cache does not need to handle misaligned accesses.

Based on the lower bits of the effective address, the second operand is shifted to the correct position for a word-wise access, resulting in the data bus $data$. In parallel the circuit **Sh4S** computes for every byte $i \in \{0, \dots, 3\}$ of the data word a usage bit ub_i which marks the bytes used by the access. Hence, for loads the bus ub_* selects the bytes to be read, for stores ub_* selects the bytes that are written.

The data cache uses the write bit, the effective address, the data bus, and the usage bits to perform the memory access. For load accesses the data cache returns the result on the data bus. Along with the result, the data cache returns the effective address of the access, the page fault interrupt signal Dpf received from the main memory, and the control signals which are passed unchanged by the cache, e.g., the tag of the instruction.

The memory unit is able to handle load word left (LWL) respectively load word right (LWR) instructions (see table A.1 in the appendix). These instructions update only parts of the target registers and are used to access misaligned words. Since the processor core updates always whole register entries, the destination register is also used as second operand. The result of the load access is combined with the content of the destination register to compute the result. Thus, for LWL/LWR instructions, the data cache must return the content of the main memory for the bytes for which ub_* is active, and the content of the second operand for the bytes for which ub_* is inactive.

The circuit **Sh4L** shifts the result of the cache access as requested by the instruction based on the lower bits of the effective address and the value of the control signals in the bus con . The result of load accesses is returned to the processor on the high and the low part of the CDB (see table 4.3 on page 63). If an interrupt occurred the memory unit returns the effective address on the low part of the CDB.

The construction and the pipelining of the circuits **Sh4S** and **Sh4L** is straightforward and not discussed in detail. The circuits can be found in the appendix D.4. Delay and cost of the circuits are:

$$\begin{aligned}
 D(\text{Sh4S}) &\leq \max\{D(\text{Add}(32)), D(\text{Add}(2)) + D(\text{HDec}(2)) + 2 \cdot D_{MUX}\}, \\
 D(\text{Sh4L}) &\leq 4 \cdot D_{MUX}, \\
 C(\text{Sh4S}) &\leq C(\text{Add}(32)) + 2 \cdot C(\text{Dec}(2)) + C(\text{HDec}(2)) \\
 &\quad + 104 \cdot C_{MUX} + 11 \cdot C_{OR} + 10 \cdot C_{AND}, \\
 C(\text{Sh4L}) &\leq C(\text{Inc}(2)) + C(\text{Dec}(2)) + C(\text{Sel}(4)) \\
 &\quad + 186 \cdot C_{MUX} + 3 \cdot D_{OR} + D_{AND}.
 \end{aligned}$$

5.2 Overview of the Data Cache

The data cache presented in this thesis is a “non-blocking write-through write-allocate” cache. In contrary to the simpler blocking caches, a *non-blocking* cache does not stall the memory unit in case of a cache miss. It can service multiple misses at a time and return the result of a hit before a preceding miss has completed. A *write-through* cache

updates the main memory (or the next cache level) for every write access to the cache. Thus, the main memory always contains the same data as the cache. In contrary the write-back variant only updates the memory if a line that has been written is evicted out of the cache. A new cache-line is written into the cache whenever an access misses the cache. This is called *write-allocate*. A read-allocate cache would only write new lines into the cache for read-misses.

A *cache-line* contains $S_{DC} = 2^{s_{DC}}$ (aligned) bytes. On a miss always a whole cache-line is read from the main memory and saved in the cache. The data cache is assumed to be non-sectored, i.e. the width of the cache RAM equals the width of a cache-line. Accesses to the cache RAM always write or read whole cache-lines. The data cache is a K_{DC} -way set associative cache ($K_{DC} = 2^{k_{DC}}$). Every line of the main memory can be saved at K_{DC} different locations in the cache. The location of a cache-line currently saved in the cache is defined by the *way* of the cache-line.

The basics of non-blocking caches were presented by Kroft [Kro81]. The presented design of a non-blocking cache is based on the work of Sicolo [Sic92]. In his thesis Sicolo gives an overview of the basic structures of a non-blocking cache, but does not handle the gate-level implementation, pipelining and interrupts. To solve these problems the design of Sicolo had to be adopted significantly.

The cache design presented by Sicolo combines two succeeding stores to the same cache-line to a single store. This is not possible for a first level cache as writes have to be executed in order due to interrupts. Also Sicolo uses a write-back strategy which has the following drawback not handled in his work: before a new cache-line is written into the cache it must be checked if any succeeding access will replace this line. If this is the case and the cache-line has been modified by a store the cache-line must not be written into the cache but back into the main memory.

Note that for the check the way in which the succeeding accesses write is needed. Hence, only those accesses can be taken into account for which the way has already been computed. To cover the remaining accesses, additionally every new access must be checked if it will overwrite a cache-line which is about to be written into the cache. This can only be done after the hit signal for the instruction has been computed and therefore delays misses.

Figure 5.2 depicts an overview of the data cache. The cache consists of four sub-circuits: the hit computation HC, the update queue UpdQ, the read queue ReadQ, and the cache core Core containing the actual cache RAM. The update queue combines the miss queue and the replace queue of the design presented in [Sic92].

The cache core contains the cache memory, the cache directory and the replacement circuit. The update queue holds all accesses that will update the cache core eventually (i.e., store instructions and cache misses). The read queue contains all read misses. The circuit HC computes the overall hit signal, taking the content of the queues and the cache core into account.

5.2.1 Execution of Memory Accesses

Depending on the type of the memory access, different data are required for the execution of the access. Loads only require the data that are actually read. Stores require the

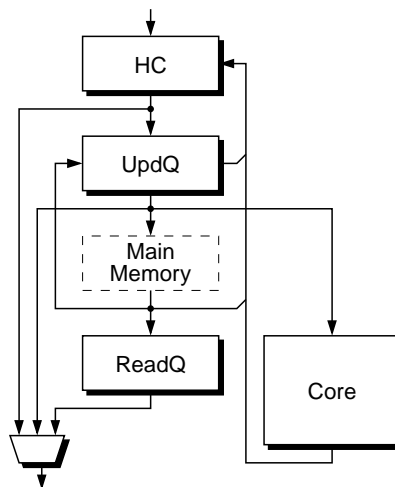


Figure 5.2: Overview of the data cache

whole cache-line in order to update the cache core. These are called the required data of the access.

New memory accesses first enter the hit computation. The hit computation computes a hit signal indicating whether the required data of the access are in the cache. It may happen that only parts of the data needed by an access are in the cache, e.g., if a load follows a store to the same address. In this case the access is treated as miss. Apart from the hit signal the hit computation also returns all required bytes of which the value is known. These bytes are marked valid using an additional valid bit per byte.

Based on the result of the hit computation the type of the access instructions are sent to different queues. Stores are sent to the update queue. Load misses are sent to both the update queue and the read queue. The load hits are directly returned to the memory unit. Hence, the data cache can return instructions out-of-order.

The misses in the update queue start read requests to the main memory. The main memory returns the result of the access on the result bus. The content of the result bus is used to update the entries in the update queue as well as the read queue. As soon as all bytes of a load in the read queue are valid, the result is returned to the memory unit. If all the bytes of a cache-line in the update queue are valid, the cache core can be updated. Since the cache is a write through cache, no cache-line must be evicted to the main memory before it is overwritten. If the entry in the update queue is a store, the main memory is also updated and the store is returned to the memory unit.

5.2.2 Cache Core and Main Memory

The handling of misses is done outside the cache core macro. Therefore the data path of the cache core can be the same as for a simple blocking cache. Additionally the cache core does not handle any data dependencies. This enables to easily pipeline the core. The only problem may be the replacement circuit, which decides which cache-line has to be overwritten. Advanced replacement algorithms (e.g., LRU) take all preceding instructions into account and assume that the instructions in the cache are handled in order. Yet a strict adherence of the replacement circuit according to

the strategy is not needed for data-consistency. Therefore, in this thesis a pipelined LRU algorithm that does forward the preceding instructions is used which simplifies the design of the cache core.

The main memory is not assumed to return the read accesses in order. This makes it possible to build multiple levels of non-blocking caches using the presented design. To identify the results returned from the main memory, the main memory must return the address of the request along with the data.

5.2.3 Speculation

If an instruction causes an interrupt or has been mispredicted, all succeeding instructions must be aborted. Updates of the cache core or the main memory are not recoverable. Thus, a store instruction may not update the cache core or the main memory as long as the instruction may be aborted. An instruction won't be aborted if the instruction does not cause a repeat interrupt (i.e., a page fault) and all preceding instructions have retired, i.e. the instruction is the oldest active instruction.

The page fault interrupt is computed by the main memory. It can only be caused by accesses to cache-lines that are not in the cache. Thus the value of the page fault interrupt is known before a store instruction updates the cache or the main memory, since the instruction first tries to fetch the cache-line from the main memory in case of a miss. If this read access causes a page fault, the store instruction is not executed.

To guarantee that store instructions won't update the cache core or the main memory before they become the oldest active instruction, two solutions are possible. The simple solution is to stall a store instruction at the end of the hit computation until it becomes the oldest active instruction. Due to the special memory reservation station (see section 4.2.2) a store instruction is always the oldest memory instruction when it enters the memory unit. Thus, no preceding instruction gets stalled and the store instruction eventually becomes the oldest active instruction.

This simple variant has some drawbacks: load instructions may not overtake store instructions in the reservation station. Thus, a store instruction stalls all succeeding load instructions. Further more, before a store instruction can leave the hit computation all queues of the cache have to be emptied. This means especially that at most one store instruction can be in the update queue.

A more efficient solution is to stall store instructions in the update queue just before they update the cache RAM or the main memory. The stores are stalled until they become the oldest instruction and it is clear that they did not cause an interrupt. This method is used in the presented design and is described in more detail in the update queue section 5.5.

5.3 Hit Computation

The hit computation computes the two signals *hit* and *sl*. The signal *hit* (called *hit signal*) is active if the required data is already in the cache. The signal *sl* (called *same-line signal*) is active if a preceding instruction inside the cache accesses the same line. In this case even if the requested data is not yet in the cache, no new request to the main memory has to be made. The signals *hit* and *sl* may be active at the same time,

e.g., if the cache-line containing the required data has been requested by a preceding instruction and has just been returned by the main memory. The hit signal has a higher priority.

If an memory access succeeds a store that writes some of the requested data of the access, it may happen that only parts of the requested data are known at the time the hit computation returns the data for the access. Therefore, the circuit HC computes a byte-valid signal bv_i for every byte i of the requested data. The signal bv_i is active if the value of the byte i is known. The hit signal hit is active if the byte-valid signals for all required bytes of the access are active. The signals hit , sl , and bv_* together are called *global hit signals*.

For every byte i for which bv_i is active, the hit computation returns the value written by the last store being processed by the cache which updates this byte. If no such instruction exists, the hit computation returns the current content of the cache RAM for this byte.

In order to later update the cache core, the hit computation also computes the bus *way* which points to the way of the cache core which has to be written. If the cache currently processes an access that addresses the same cache-line, the way must be the way computed for this access. Otherwise the value computed by the replacement circuit of the cache core is used. Note that in case of a hit the replacement circuit returns the index of the way that contains the requested data.

Based on the type of the access and the value of the global hit signals, the hit computation computes the action to be done for the instruction. The instruction can be sent to the update queue, the read queue, both queues, or can directly be returned to the memory unit.

Note that an access is treated as hit whenever the requested data can be found in the cache, even if it would be a miss if the accesses would be handled sequentially. Assume the cache core contains a cache-line l_0 and an access to the cache-line l_1 will replace the cache-line l_0 . If a second access to the cache-line l_0 is started before the line is replaced, this access is treated as hit as the cache core returns the correct data. If the access to the cache-line l_0 is a store, it will update the cache core and thereby overwrite the cache-line l_1 written into the cache core by the preceding access. Since the cache is a write-through cache this can be done without checking if data would be lost.

5.3.1 Overview of the Hit Signal Computation

The requested data of an instruction can be located in four different places in the cache: in the cache core, in the update queue, on the result bus of the main memory, or as write data of a preceding store in the hit computation itself. If the requested data cannot be found in any of these places, the data must be loaded from the main memory.

The update queue and the hit computation compute for every entry respectively pipeline stage a local same-line signal and local byte-valid signals. The local same-line signal is active if the entry or stage contains an instruction which accesses the same line as the instruction for which the global hit signals are being computed. Let $S_{DC} = 2^{s_{DC}}$ be the number of bytes of a cache-line. Then two instructions access the same lines if the bits 31 to s_{DC} of the addresses (called *line-address*) are equal. The

local byte-valid signals indicate that the entry or stage contains the correct value for that byte. They may only be active if the local same-line signal is active. For the cache core and the result bus of the main memory all bytes are valid on a hit. Therefore, these circuits only compute a local hit signal which is active if the requested line is found.

The global hit signals must reflect the content of the cache at the time the queues are updated. The computation of the global hit signals may take multiple cycles due to pipelining. Therefore, all changes to the content of the cache during the computation must be forwarded in order to be reflected in the result. The cache core is only updated by the update queue. Thus, in order to forward all possible changes of the cache core it suffices to forward the content of the update queue at the time the hit computation is started and all updates to the update queue.

The entries of the update queue are updated by the result bus of the main memory and by the hit computation for allocating new entries. Note that stores are processed in order. Thus, only the instructions in the hit computation which are started before the instruction for which the global hit signals are being computed have to be taken into account. Hence, all possible updates to the update queue by the hit computation are known at the time the hit computation for an access is started. The updates to the update queue by the main memory are not known at the time the hit computation is started. They must be forwarded in every cycle of the hit computation.

Figure 5.3 shows an overview of the computation of the global hit signals. For the computation of the global hit signals, the hit computation first computes *static* and *dynamic hit signals*. The static hit signals are based on the content of the hit computation, the update queue, and the cache core. They must represent the value of these circuits at the time an instruction enters the hit computation and have to be computed only once. The dynamic hit signals are based on the content of the result bus of the main memory and have to be computed in every cycle. Since all bytes of the result bus are valid, the dynamic hit signals only consist of a hit signal (the dynamic hit signal) without byte-valid signals.

The dynamic hit signal must be set if the line-addresses of the current data on the result bus and the instruction for which the hit signals are being computed are equal. Assume the cache is a RAM block with $32 - s_{DC}$ address bits that is written by the result bus of the main memory. If the computation of the global hit signal is seen as read to that RAM block the dynamic hit signal must be set exactly if the write port would be forwarded to the read port of the RAM block. Thus, the computation of the dynamic hit signals can be done using a forwarding circuit (with stalling) of such a RAM with $32 - s_{DC}$ address bits.

Let c_{M2H} denote the number of cycles needed for the computation of the global hit signals based on the content of the result bus of the main memory. In order to take all data from the result bus into account that have updated the update queue at the time the global hit signals are computed, the result bus is delayed by $c_{M2H} - 1$ cycles similar to the pipelined forwarding circuits (see section 2.6.3). The forwarding circuit then uses the un-delayed result bus and the $c_{M2H} - 1$ additional stages of the result bus are forwarded using a forwarding tree. This tree is included into the computation of the static hit signals.

The circuit **staticHC** in figure 5.3 computes the static hit signals *ssl* (static same-line) and *sbv_{*}* (static byte-valid) along with the corresponding data busses *sbyte_{*}* and

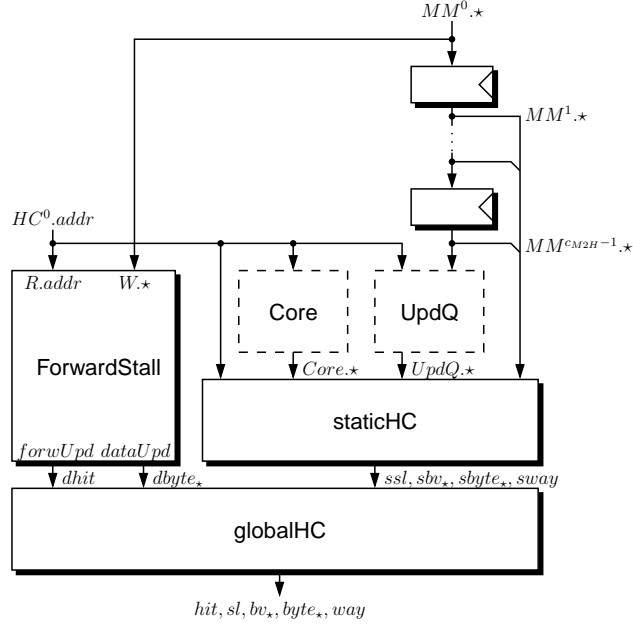


Figure 5.3: Overview of the hit signal computation

the way *sway*. For this computation, the cache core and the update queue are accessed to compute local hit signals. Additionally to these local hit signals the circuit **staticHC** uses the stages 1 to $c_{M2H} - 1$ of the delayed result bus and the preceding instructions in the hit computation to compute the static hit signals.

The dynamic hit signal *dhit* and the corresponding data busses *dbyte** are computed using a forwarding circuit with stalling (**ForwardStall**). This circuit uses the un-delayed result bus. The outputs of the two circuits **staticHC** and **ForwardStall** are combined in the circuit **globalHC** to compute the global hit signals, byte values and the way.

5.3.2 Local Hit Signals

For the computation of the static hit signals, first local same-line and byte-valid signals must be computed for all stages of the hit computation and all update queue entries. These signals are called $HC.lsl_i$ (local same-line) and $HC.lbv_{*,i}$ (local byte-valid) for stage i of the hit computation and $UpdQ.lsl_i$ respectively $UpdQ.lbv_{*,i}$ for the entry i of the update queue. For the stages of the delayed result bus of the main memory and the cache core a local hit signal must be computed. These signals are called $MM.lhit_i$ (local hit) for the stage i of the delayed result bus, and $Core.lhit$ for the cache core.

The local same-line signal indicates that the stage or entry holds an access which addresses the same cache-line as the access for which the hit signals are being computed. Which bytes of this cache-line are valid is defined by the local byte-valid signals. The local hit signals for the result bus and cache core are active if the required data is in the delayed result bus stage respectively cache core. The computation of the local signals for cache core and update queue are described in the section 5.4 and 5.5.

To compute the local hit signals for the hit computation stages, the bus ub_* indicating which bytes of a word are used must be extended to a whole line. Figure 5.4

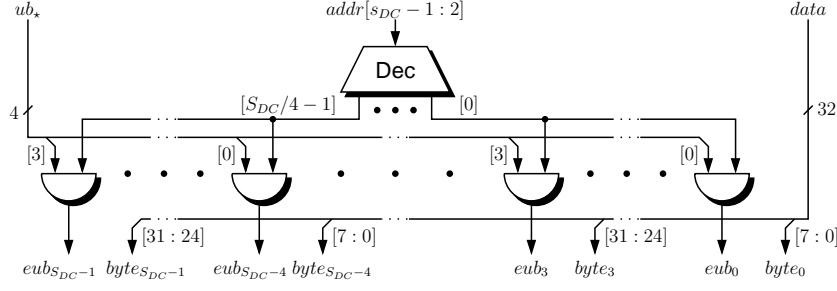


Figure 5.4: Computation of the signals eub_* and $byte_*$

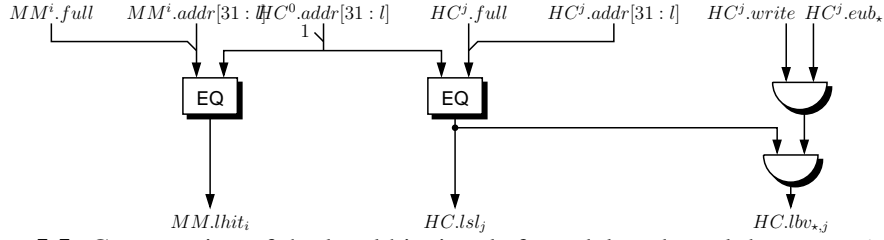


Figure 5.5: Computation of the local hit signals for a delayed result bus stage i and a hit computation stage j

depicts the computation of the extended usage-bits eub_* . The byte $4 \cdot m + n$ for $m \in \{0, \dots, L/4 - 1\}$ and $n \in \{0, \dots, 3\}$ of a line is used if the instruction accesses the word m of the line and the byte n of the word is used. Which word of a cache-line is accessed can be computed by decoding the bits $s_{DC} - 1$ to 2 of the address. Additionally the circuit in figure 5.4 computes the byte busses $byte_{s_{DC}-1 \dots 0}$ by copying the data bus $S_{DC}/4$ times. The computation of the signals eub_* and $byte_*$ must only be done once for every instruction which enters the hit computation.

The circuit in figure 5.5 computes the local hit signals from the accesses in the hit computation and the stages of the delayed result bus. Let c_{HC} be the number of cycles of the hit computation. Let I be an instruction which enters the hit computation, i.e., I is in stage 0. An instruction in stage $j \in \{1, \dots, c_{HC} - 1\}$ of the hit computation accesses the same line as I if the line address $HC^j.addr$ of the instruction in stage j is the equal to the line address $HC^0.addr$ of the instruction I in stage 0. If this is the case and the stage j contains a valid access ($HC^j.full = 1$) the signal $HC.lsl_j$ is activated. If $HC.lsl_j$ is active, the local byte-valid signals are activated for all bytes written by the instruction in stage j (indicated by $HC^j.eub_* \wedge HC^j.write$). Note that this disables all byte-valid signals if the access in stage j is a load.

The local hit signal for stage $i \in \{1, \dots, c_{M2H} - 1\}$ of the delayed result bus can be computed by comparing the line addresses of the delayed result bus stage $MM^i.addr$ with the address of I and checking the full bit $MM^i.full$. Since all bytes of the result bus are valid no byte-valid signals are needed.

5.3.3 Static Hit Signals

The static byte-valid signal for a byte k sbv_k must be active if any local byte-valid signal for the byte k of a hit computation stage or an update queue entry is active or the local hit signal for a delayed result bus stage or the cache core is active. If the

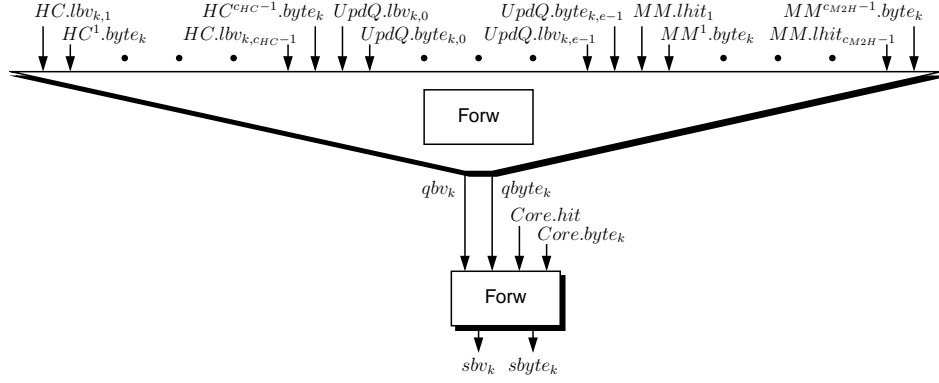


Figure 5.6: Computation of the signals sbv_j and $sbyte_j$ for a byte j

signals sbv_k is active, the bus $sbyte_k$ for the corresponding data must contain the value of the last instruction preceding to I which updates that byte. If no such instruction exist, the bus $sbyte_k$ must contain the current content of the main memory. Figure 5.6 shows the computation of the signals sbv_k and $sbyte_k$ for a byte k .

The computation of the byte-valid signals and the corresponding bytes can be done using a tree as in the forwarding tree (see section 2.6.3). The tree computes an OR of the local byte-valid signals (respectively hit signals for cache core and result bus). Parallely the tree selects the first byte (from the left) for which the byte-valid signal is active.

The outputs of the cache core $Core.lhit$ and $Core.byte_*$ are assumed to be timing critical as they depend on an access to the cache RAMs. Therefore, they are not used before the last stage of the tree. The outputs of the tree which do not take the signals from the cache core into account are called qbv_* respectively $qbyte_*$. Thus, for a byte k with $0 \leq k \leq S_{DC} - 1$ holds:

$$sbv_k = qbv_k \vee Core.hit, \quad (5.1)$$

$$sbyte_k = \begin{cases} qbyte_k & \text{if } qbv_k = 1 \\ Core.byte_k & \text{if } qbv_k = 0 \end{cases}. \quad (5.2)$$

The instructions in the hit computation have been started after the instruction in the update queue and must have a higher priority in the tree. Since instructions enter hit computation and update queue in order and are not reordered within, the stages respectively entries with lower index must have higher priority. The result bus of the main memory and the cache core only contain data written by already completed instructions and therefore have a lower priority than hit computation and update queue. Since cache and main memory contain the same data due to the write-through strategy, the order is irrelevant for the result bus stages and the cache core.

The static same-line signal ssl must be active if the local same-line signal is active for any hit computation stage or update queue entry. The static *way* must be the way of the last instruction for which the local same-line signal is active. If none of the same-line signals is active, the way must be set to the way output of the core $Core.way$. The circuit for the computation of the signals ssl and $sway$ is shown in figure 5.7.

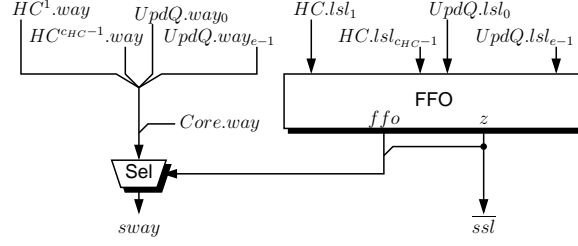


Figure 5.7: Computation of the signals *ssl* and *sway*

Among other things, the static way of the instruction in stage 0 of the hit computation *sway* depends on the way of the instruction in stage 1 which is computed only one cycle before *sway*. Thus, the delay of the path $HC^1.way \rightsquigarrow sway$ must be at most one cycle. In order to minimize the delay on this path, the static way is not computed using a forward tree like the byte-valid signals and the data. Instead, the first instruction for which the local same-line signal is active is computed using a find-first-one circuit. If none of the local same-line signals is active the zero output z of the find-first-one circuit is active. Thus, the static same-line signal must be active, if the zero output is inactive. Using the output *ffo* which unary selects the first instruction with active same-line signal, the way of this instruction is selected. If the zero output is active, the way of the core is selected. In order to minimize the delay for the signals $HC^1.way$ and *Core.way*, the select circuit is “unbalanced” accordingly.

5.3.4 Global Hit Signals

The global byte-valid signal for a byte k bv_k has to be active if either the static byte-valid signal for this byte sbv_k or the dynamic hit signal *dhit* is active. Since only the processor writes the main memory, the dynamic data $dbyte_\star$ from the main memory cannot be newer than the static data $sbyte_\star$ in the cache. Thus, if the static byte-valid signal is active, the static data have to be returned:

$$bv_k = sbv_k \vee dhit, \quad (5.3)$$

$$byte_k = \begin{cases} sbyte_k & \text{if } sbv_k = 1 \\ dbyte_k & \text{if } sbv_k = 0 \end{cases}. \quad (5.4)$$

The global hit signal *hit* must be active if all bytes required by the instruction are valid. For write instructions all bytes of the cache line are required in order to update the cache core. Yet the bytes of the line written by the instruction (indicated by $HC^0.eub_\star$) are valid and need not to be found in the cache. For read instructions the bytes to be read (indicated by $HC^0.eub_\star$) need to be valid:

$$hit = \bigwedge_{k=0}^{S_{DC}-1} \left(bv_k \vee \overline{(HC^0.eub_k \oplus HC^0.write)} \right).$$

Note that the signal $HC^0.eub_\star$ and $HC^0.write$ are known early in the hit computation and not as timing critical as the dynamic hit signal and the hit signal from the cache core. Therefore, the computation of the hit signal is optimized using the

distributive law:

$$\begin{aligned}
 hit &\stackrel{(5.3)}{=} \bigwedge_{k=0}^{S_{DC}-1} \left(dhit \vee sbv_k \vee \overline{(HC^0.eub_k \oplus HC^0.write)} \right) \\
 &\stackrel{(5.1)}{=} \bigwedge_{k=0}^{S_{DC}-1} \left(dhit \vee Core.hit \vee qbv_k \vee \overline{(HC^0.eub_k \oplus HC^0.write)} \right) \\
 &= dhit \vee Core.hit \vee \bigwedge_{k=0}^{S_{DC}-1} \left(qbv_k \vee \overline{(HC^0.eub_k \oplus HC^0.write)} \right).
 \end{aligned}$$

No dynamic way and same-line signal are computed for the result bus of the main memory. The static way and same-line signal are therefore used as global way and same-line signal:

$$\begin{aligned}
 sl &= ssl \\
 way &= sway
 \end{aligned}$$

5.3.5 Actions

Based on the global hit signals the hit computation either directly returns the result to the memory unit or allocates new entries in the read and the update queue. If a new update queue entry is allocated, also the bits *req* and *rdy* for this entry have to be computed. The request bit *req* has to be active if the entry must request the line from the main memory. The ready bit *rdy* indicates that the entry contains already the correct data. Based on the signals *hit* and *miss* and the type of the access (indicated by the signal *write* which is active for stores), the following actions have to be taken:

- $\overline{write} \wedge hit$: The instruction is a load-hit. The result of the load can be returned from the hit computation directly to the memory unit.
- $\overline{write} \wedge \overline{hit} \wedge sl$: The instruction is a load-miss but a preceding instruction accesses the same line. A new entry in the read queue has to be made. Since the required data will be requested by a preceding instruction no new update queue entry is needed.
- $\overline{write} \wedge \overline{hit} \wedge \overline{sl}$: The instruction is a load-miss and no preceding instruction accesses the same line. New entries in the read queue and the update queue have to be made. The request bit *req* of the new entry is set to one. The ready bit *rdy* of the entry is set to zero.
- $write \wedge hit$: The instruction is a store-hit. A new entry in the update queue has to be made. The bit *rdy* is set to one and the bit *req* is set to zero.
- $write \wedge \overline{hit} \wedge sl$: The instruction is a store-miss but a preceding instruction accesses the same line. A new entry in the update queue has to be made. Since no new read request for the cache-line has to be made, the bits *rdy* and *req* are set to zero.

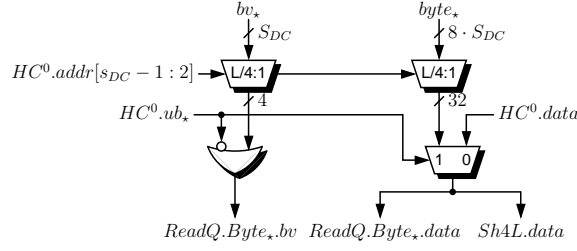


Figure 5.8: Computation of the data bus to the read queue and the memory unit

- $write \wedge \overline{hit} \wedge \overline{sl}$: The instruction is a store-miss and no preceding instruction accesses the same line. A new entry in the update queue has to be made, which requests the required cache-line from the main memory. The bit rdy is set to zero and the bit req is set to one.

The result of a load access is returned to the memory unit using the bus $Sh4L.*$. New entries for the update queue and the read queue are allocated using the busses $UpdQ.*$ respectively $ReadQ.*$. The full bit of the busses is active if the corresponding action has to be taken. Based on the preceding list the full bits to the circuit **Sh4L** and the two queues as well as the bits req and rdy for the bus to the update queue can be computed as follows:

$$\begin{aligned}
 Sh4L.full &= \overline{write} \wedge hit, \\
 ReadQ.full &= \overline{write} \wedge \overline{hit}, \\
 UpdQ.full &= write \vee (\overline{hit} \wedge \overline{sl}), \\
 UpdQ.rdy &= hit, \\
 UpdQ.req &= \overline{hit} \wedge \overline{sl}.
 \end{aligned} \tag{5.5}$$

The byte-valid signals and the corresponding data sent to the update queue are called $UpdQ.Byte_*.bv$ and $UpdQ.Byte_*.data$. The valid signals for the bytes of the update queue must be active for all bytes for which the byte-valid signals of the hit computation are active or which are written by the instruction. The byte data for the update queue are the global byte values. For a byte k with $0 \leq k \leq S_{DC} - 1$ holds:

$$\begin{aligned}
 UpdQ.Byte_k.bv &= (HC^0.eub_k \wedge HC^0.write) \vee bv_k, \\
 UpdQ.Byte_k.data &= \begin{cases} HC^0.byte_k & \text{if } HC^0.eub_k = 1 \\ byte_k & \text{if } HC^0.eub_k = 0 \end{cases}.
 \end{aligned}$$

The data bus $ReadQ.Byte_*.data$ to the read queue is only 32 bits wide. Thus, the requested word must be selected out of the cache-line returned by the global hit computation. Let W be the index of the requested word in the cache-line, i.e.:

$$W := \langle HC^0.addr[s_{DC} - 1 : 2] \rangle.$$

Then, the bytes W to $W + 3$ have to be selected in order to compute the data bus to the read queue. The byte-valid signals for the read queue $ReadQ.Byte_*.bv$ must be active for all bytes of the word which are valid or not used by the access (indicated by

the un-extended usage bits $HC^0.ub$). Due to the LWL and LWR instructions the bytes which are not used by the instruction have to be set to the value of the value of the input data bus of the instruction. Hence, the busses $ReadQ.Byte_\star.\{bv, data\}$ to the read queue can be computed as ($0 \leq k \leq 3$):

$$ReadQ.Byte_k.bv = \overline{HC^0.ub_k} \vee bv_{W+k},$$

$$ReadQ.Byte_k.data = \begin{cases} HC^0.data[8 \cdot k + 7 : 8 \cdot k] & \text{if } HC^0.ub_k = 1 \\ byte_{W+k} & \text{if } HC^0.ub_k = 0 \end{cases}.$$

The data on the bus $ReadQ.Byte_\star.data$ contains all requested data in case of a cache-hit. Thus, the data bus to the memory unit $Sh4L.data$ can be directly derived from the bus $ReadQ.Byte_\star.data$. The bus to the memory unit does not need byte-valid signals, since all bytes have to be valid for read hits. Figure 5.8 depicts the computation of the data busses and the byte-valid signals to the read queue and the data bus to the memory unit.

5.3.6 Stall Computation

The hit computation has to be stalled if the action for the access cannot as described above be performed. A result cannot be returned to the memory unit if $Sh4L.stallOut$ is active. A new read or update queue entry cannot be made if the corresponding stall signals $ReadQ.stallOut$ or $UpdQ.stallOut$ are active. Hence, the input stall signal for the hit computation can be computed as:

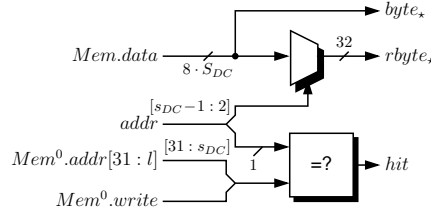
$$HC.stallIn = (Sh4L.full \wedge Sh4L.stallOut) \\ \vee (ReadQ.full \wedge ReadQ.stallOut) \\ \vee (UpdQ.full \wedge UpdQ.stallOut).$$

5.3.7 Cost and Delay

The overall delay of the hit computation is dominated by the access to the cache RAM in the cache core. Thus, the critical signals are $Core.hit$ and $Core.byte_\star$. As one can easily verify the output of the hit computation with the highest delay is the data bus to the read queue $ReadQ.Byte_\star.data$. Thus, the overall delay of the hit computation is:

$$D(HC) \leq D(\mathbf{Core}) \quad (Core.byte_\star) \\ + D_{MUX} \quad (sbyte_\star, \text{equation (5.2)}) \\ + D_{MUX} \quad (byte_\star, \text{equation (5.4)}) \\ + (s_{DC} - 2) \cdot D_{MUX} + D_{MUX}. \quad (ReadQ.Byte_\star.data, \text{figure 5.8})$$

To compute the number of cycles the result bus of the main memory has to be delayed, the maximum delay from the first stage of the result bus $MM^0.\star$ to the outputs of the hit computation needs to be known. As above, the path from to the bus $ReadQ.Byte_\star.data$ has the highest delay out of all paths that start at $MM^0.\star$. The

**Figure 5.9:** Modified test circuit

delay this path is:

$$\begin{aligned}
 D(MM^0.\star \rightsquigarrow ReadQ.Byte_\star.data) &\leq D(EQ(33 - s_{DC})) + D_{MUX} \\
 &\quad (dbyte_\star, \text{figure 5.3}) \\
 &\quad + D_{MUX} \quad (byte_\star, \text{equation (5.4)}) \\
 &\quad + (s_{DC} - 2) \cdot D_{MUX} + D_{MUX}.
 \end{aligned}$$

This delay can be reduced, if the selection step from the cache-line to the accessed word in figure 5.8 is removed from the critical path. The selection for the cache-lines of the result bus can be done in parallel to the computation of the forward signal using a modified circuit **Test** (see figure 5.9) in the forward tree. The circuit computes a second data output $rbyte_\star$ which contains the data needed by read accesses. This output is treated like the standard data in the forwarding circuit, i.e. the forwarding circuit computes $8 \cdot S_{DC} + 32$ data bits. The additional byte busses computed by the forwarding circuit are called $drbyte_\star$.

Using the new bus $drbyte_\star$ the output to the read queue can be computed as follows (for a byte k with $0 \leq k \leq 3$ and $B := 8 \cdot k$):

$$\begin{aligned}
 ReadQ.Byte_k.data &= \begin{cases} HC^0.data[B + 7 : B] & \text{if } HC^0.ub_k = 1 \\ byte_{W+k} & \text{if } HC^0.ub_k = 0 \end{cases} \\
 &\stackrel{(5.4)}{=} \begin{cases} HC^0.data[B + 7 : B] & \text{if } HC^0.ub_k = 1 \\ sbyte_{W+k} & \text{if } HC^0.ub_k = 0 \wedge sbv_{W+k} = 1 \\ dbyte_{W+k} & \text{else} \end{cases} \\
 &= \begin{cases} HC^0.data[B + 7 : B] & \text{if } HC^0.ub_k = 1 \\ sbyte_{W+k} & \text{if } HC^0.ub_k = 0 \wedge sbv_{W+k} = 1 \\ drbyte_k & \text{else} \end{cases} \\
 &\stackrel{(5.2)}{=} \begin{cases} HC^0.data[B + 7 : B] & \text{if } HC^0.ub_k = 1 \\ qbyte_{W+k} & \text{if } HC^0.ub_k = 0 \wedge qbv_{W+k} = 1 \\ Core.byte_{W+k} & \text{if } HC^0.ub_k = 0 \wedge qbv_{W+k} = 0 \\ drbyte_k & \text{else} \end{cases} \quad \text{if } \wedge Core.hit = 1
 \end{aligned}$$

Note that the most critical bus on which the bus $ReadQ.Byte_\star.data$ depends is $drbyte_\star$ since it determines the delay of the path from the main memory to the read queue and therefore determines the number of cycles the result bus has to be delayed. The second

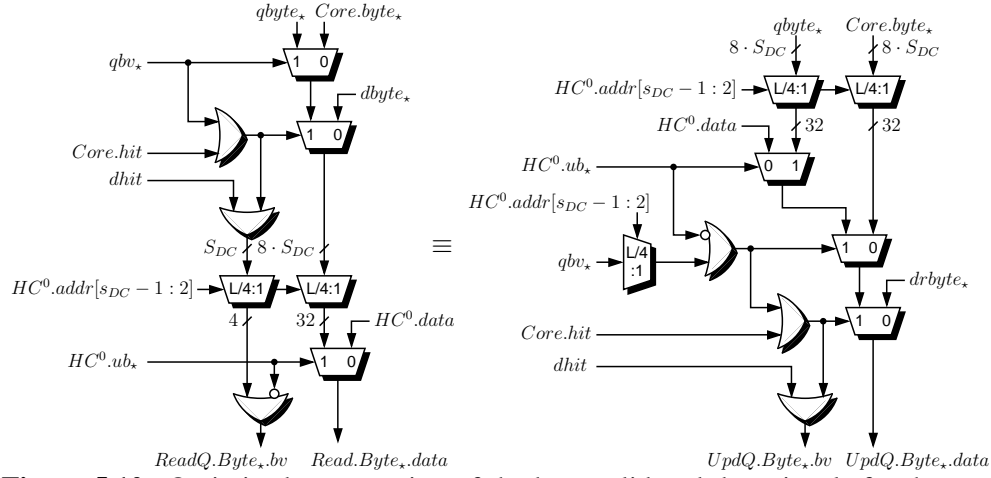


Figure 5.10: Optimized computation of the byte-valid and data signals for the read queue

most critical signals are the outputs of the cache core $Core.\star$ since they determine the overall delay of the hit computation.

Figure 5.10 depicts how the order of the multiplexers which compute the bus $ReadQ.Byte_\star.data$ can be changed in order to use the critical signals as late in the multiplexer-tree as possible. The left side of the figure depicts the computation of the signals $ReadQ.Byte_\star.\{data, bv\}$ as described in the previous sections. This circuit can be transformed into the circuit on the right side of the figure by moving the multiplexers for the signals $Core.byte_\star$ and $dbyte_\star$ to the bottom (and replacing $dbyte_\star$ by $drbyte_\star$). Note that this also reduces the delay of the signals $ReadQ.Byte_\star.bv$ that also depends on the output of the cache core and the memory unit (via the signals $Core.hit$ respectively $dhit$). The modifications depicted in figure can also be used in order to reduce the delay of the busses to the update queue $UpdQ.Byte_\star.\{data, bv\}$.

Finally the delay of the reduction from the cache-line to the accessed word can be reduced if the address is decoded and a unary select circuit is used. Figure 5.11 shows a version of the global hit computation which takes all optimizations into account.

Using the circuit in figure 5.11 the overall delay of the hit computation is:

$$D(HC) \leq D(\text{Core}) + D(\text{Sel}(S_{DC}/4)) + 2 \cdot D_{MUX}. \quad (5.6)$$

The delay of the path from the result bus to outputs of the hit computation are:

$$\begin{aligned} D(MM^0.\star \rightsquigarrow ReadQ.Byte_\star.data) \leq & \max\{D(\text{EQ}(33 - s_{DC})), \\ & (s_{DC} - 2) \cdot D_{MUX}\} \\ & (hit, rbyte_\star, \text{figure 5.9}) \\ & + D_{MUX} \quad (drbyte_\star, \text{figure 5.3}) \\ & + D_{MUX}. \end{aligned}$$

Let c_{HC} be the number of cycles needed for the hit computation, and let c_{M2H} be the number of cycles needed to compute the outputs of the hit computation based on the

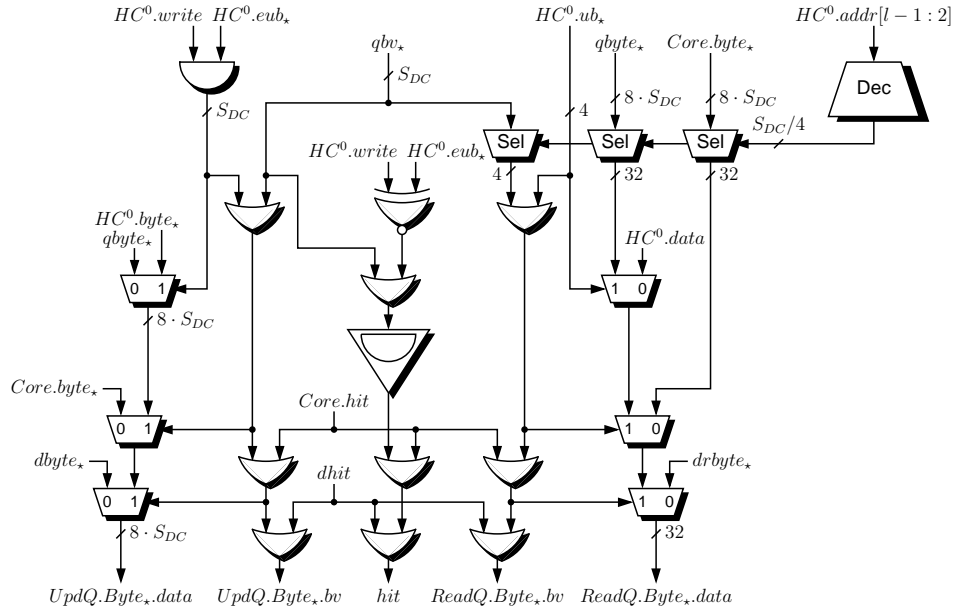


Figure 5.11: Delay optimized hit computation

result bus. Then:

$$c_{HC} = \lceil D(HC)/\delta \rceil,$$

$$c_{M2H} = \lceil D(MM.\star \rightsquigarrow ReadQ.Byte_*.data)/\delta \rceil.$$

The result bus must be delayed by $c_{M2H} - 1$ cycles.

The delay of the hit computation depends on the width of the cache-lines S_{DC} . It can be reduced without changing S_{DC} by using a sectored cache. In a sectored cache, the width of the cache-lines are a multiple of the width of the data RAM and the data bus to the memory unit. Assume $w \geq 32$ is the width of the data RAM and $S_{DC} = k \cdot w$. The cache core returns a w bit wide data bus and the delay of the hit computation is reduced to:

$$D(HC) \leq D(\text{Core}) + D(\text{Sel}(w/4)) + 2 \cdot D_{MUX}.$$

The disadvantage of a sectored cache is that it takes k cycles to return the result of a read request from the main memory or to write a cache-line into the cache core. The details of the necessary modifications are not discussed in this thesis.

Let $K_{DC} = 2^{k_{DC}}$ be the associativity of the cache core. Thus k_{DC} is the width of the bus way . Let e_{UQ} be the number of entries of the update queue. The cost of the static hit computation is (excluding registers):

$$\begin{aligned} C(\text{staticHC}) \leq & C(\text{Dec}(s_{DC} - 2)) + (S_{DC} + 1) \cdot C_{AND} \\ & + (c_{HC} - 1 + c_{M2H} - 1) \cdot C(\text{EQ}(33 - s_{DC})) + (c_{HC} - 1) \cdot C_{AND} \\ & + C(\text{ForwardTree}(32 - s_{DC}, 8 \cdot S_{DC}, c_{HC} - 1 + e_{UQ} + c_{M2H}, c_{HC})) \\ & + C(\text{FFO}(c_{HC} - 1 + e_{UQ})) + k_{DC} \cdot C(\text{Sel}(c_{HC} + e_{UQ})). \end{aligned}$$

The cost for the global hit computation is (excluding registers):

$$\begin{aligned}
 C(\text{globalHC}) \leq & 3 \cdot C_{AND} + C_{OR} + S_{DC} \cdot (C_{AND} + C_{XNOR}) \\
 & + C(\text{Dec}(s_{DC} - 2)) + 68 \cdot C(\text{Sel}(S_{DC}/4)) \\
 & + C(\text{AND-Tree}(S_{DC})) + (2 \cdot S_{DC} + 10) \cdot C_{OR} \\
 & + 3 \cdot (S_{DC} + 32) \cdot C_{MUX}.
 \end{aligned}$$

The hit computation has $32 + 32 + 4 + 1 + 1$ inputs from the memory unit, $8 \cdot S_{DC} + k_{DC} + 1$ inputs from the cache core, $e_{UQ} \cdot (9 \cdot S_{DC} + 1)$ inputs from the update queue, and $8 \cdot S_{DC} + 32$ inputs from the result bus. The hit computation computes $32 + 3$ signals sent to the memory unit, $9 \cdot S_{DC} + 5$ signals sent to the update queue, and $32 + 4 + 3$ signals sent to the read queue. The total cost for the hit computation (including all registers) is thus estimated as:

$$\begin{aligned}
 C(\text{HC}) \leq & C(\text{staticHC}) + C(\text{globalHC}) \\
 & + C(\text{ForwardStall}(32 - s_{DC}, 8 \cdot S_{DC} + 32, c_{HC}, c_{M2H})) \\
 & + (c_{HC} - 1) \cdot \lceil (25 \cdot S_{DC} + k_{DC} + 182 + e_{UQ} \cdot (9 \cdot S_{DC} + 1)) / 2 \rceil \cdot C_{REG}.
 \end{aligned}$$

5.4 Cache Core

For the cache core the same hardware can be used as for simple blocking cache (excluding the control and the interface to the main memory). The cache core is basically a k way set associative cache. The design of the cache core is not discussed in this thesis. It can be found, e.g., in [MP00]. The cache core must follow the behavior described below.

The cache core has a read and a write port which are used by the hit computation for reading and the update queue for writing. On a read access the cache core returns additionally to the data a hit signal and a way. The hit signal *hit* is active if the requested line is in the cache. Then the data output *data* contains the requested line and the way output *way* points to the way of the cache where the line is saved. If *hit* is inactive, *way* points to the way in which the requested line should be written. This is determined from the internal replacement algorithm of the cache core. For write accesses the address bus *addr* must contain the address of the line which is written and the input bus *way* must point to the way in which the line is written. The bus *data* must contain the current value of the line.

For correctness the replacement algorithm is irrelevant, e.g., a least recently used (LRU) algorithm can be used. It must only be made sure that a line is stored in at most one way. Note that multiple accesses to the cache can be made between the computation of the way for an access and the update of the cache by this access. Thus, for an LRU-algorithm at the time an access writes to the cache core the cache-line that is overwritten may not longer be the least recently used. For this to happen several accesses to addresses that are stored in the same line of the cache core must be processed at the same time. Since this is considered rare and the replacement algorithm does not affect the correctness it is acceptable.

Let $S_{DC} = 2^{s_{DC}}$ be the number of bytes of a line, $L_{DC} = 2^{l_{DC}}$ be the number of lines, and $K_{DC} = 2^{k_{DC}}$ be the number of ways of the data cache. Let c_{HC} be the

number of cycles of the hit computation. If a LRU algorithm is used, the cost for the replacement circuit **Core-Replace** with $34 - s_{DC} + k_{DC}$ inputs and k_{DC} outputs are (formulas taken from [MP00]):

$$\begin{aligned} C(\text{Core-Replace}) \leq & C(\text{RAM}(L_{DC}, K_{DC} \cdot k_{DC}, 1, 1, c_{HC})) \\ & + K_{DC} \cdot (C(\text{EQ}(k_{DC})) + C_{AND}) + C(\text{PP-OR}(K_{DC})) \\ & + K_{DC} \cdot k_{DC} \cdot 5 \cdot C_{MUX} + C(\text{Enc}(K_{DC})) \\ & + \lceil 34 - l_{DC} + 2 \cdot k_{DC}/2 \rceil \cdot c_{HC} \cdot C_{REG}. \end{aligned}$$

The number of inputs to the cache core are $33 + k_{DC} + S_{DC} \cdot 8$, the number of outputs of the cache core are $1 + k_{DC} + S_{DC} \cdot 8$. The cost and the delay of the cache core are:

$$\begin{aligned} D(\text{Core}) \leq & \max\{D(\text{RAM}(L_{DC}, 8 \cdot S_{DC}, 1, 1), \\ & D(\text{RAM}(L_{DC}, 33 - l_{DC} - s_{DC}, 1, 1)) + D(\text{EQ}(33 - l_{DC} - s_{DC})))\} \\ & + D(\text{Sel}(K_{DC})), \\ C(\text{Core}) \leq & K_{DC} \cdot (C(\text{RAM}(L_{DC}, 8 \cdot S_{DC}, 1, 1, c_{HC})) \\ & + C(\text{RAM}(L_{DC}, 33 - l_{DC} - s_{DC}, 1, 1, c_{HC})) \\ & + C(\text{EQ}(33 - l_{DC} - s_{DC}))) \\ & + 8 \cdot S_{DC} \cdot C(\text{Sel}(K_{DC})) + C(\text{OR-Tree}(K_{DC})) \\ & + C(\text{Dec}(k_{DC})) + K_{DC} \cdot C_{AND} \\ & + \begin{cases} 0 & \text{if } K_{DC} = 1 \\ C(\text{Core-Replace}) & \text{if } K_{DC} > 1 \end{cases}. \end{aligned}$$

5.5 Update Queue

The update queue contains all cache accesses that will update the cache RAM eventually, i.e., load misses and stores. A store access may only update the cache RAM or the main memory if all preceding instructions have retired and no page fault interrupt occurred for the instruction. This is necessary for precise interrupts and branch speculation.

Figure 5.12 depicts the update queue with e_{UQ} entries. The update queue is built similar to the reservation stations (see section 4.2). The update queue is controlled by the circuit **UpdQ-Control**. The circuits **UpdQ-Entry_{*}** form a queue and contain the update queue entries. New instructions are always filled into the entry 0 in the circuit **UpdQ-Entry₀**. An instruction proceeds to the next entry whenever this entry is empty.

If the update queue is full, the control raises the signal *stallOut*. This prevents the hit computation from filling new entries into the update queue. New entries are filled into the update queue by the hit computation on the bus $HC.UpdQ.*$.

The control circuit starts all read or write accesses to the main memory using the bus $UpdQ.MM.*$. The main memory can accept new requests, if the control signal $MM.ack$ is active. The result of the read accesses are sent to all queue entries using the result bus of the main memory $MM.*$ similar to the CDB in the reservation stations. If the result bus is delayed, the last stage of the delayed result bus is used by the update queue (see figure 5.3 in section 5.3).

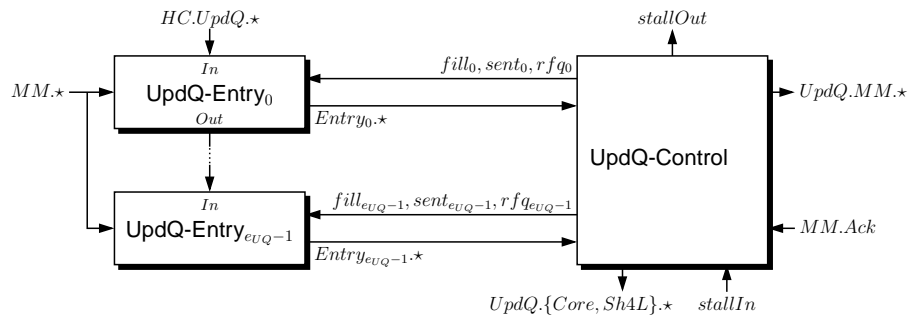


Figure 5.12: Update queue

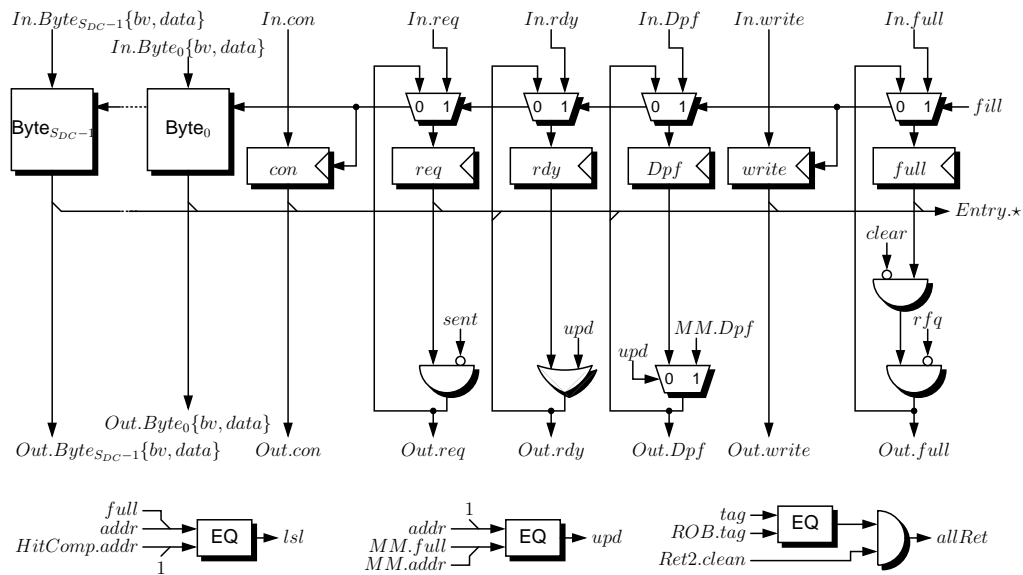


Figure 5.13: Update queue entry

The update queue updates the cache core using the bus *UpdQ.Core.★* and returns the stores to the memory unit on the bus *UpdQ.Sh4L.★*. Stores need to be returned to the memory unit in order to be removed from the reorder buffer. Note that stores must return the effective address in case of an interrupt and hence produce a result in this case. This result then has to be written into the reorder buffer via the CDB. If the memory unit cannot accept an instruction, the signal *stallIn* is raised.

5.5.1 Entries

Figure 5.13 depicts the circuit for the update queue entries. The basic structure equals the reservation station entries. If the signal *fill* is inactive the updated values of the instruction currently stored in the entry are written into the registers. If the signal *fill* is active the registers are overwritten with the updated values of the instruction in the preceding entry (respectively the hit computation for the first entry).

For every byte $k \in \{0, \dots, S_{DC} - 1\}$ of a cache-line, the update queue entries have a sub-circuit **Byte_k**. These sub-circuits store for each byte a valid bit and the data similar to the operands of the reservation station entries. Additionally each entry contains the following registers:

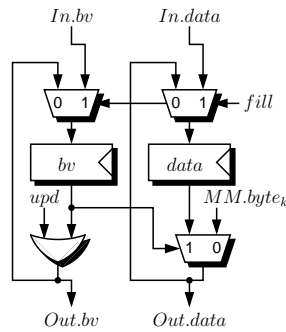


Figure 5.14: Update queue entry byte k

- *full*: This bit indicates that the entry contains a valid instruction. It is reset if the processor is flushed (indicated by the global control signal *clear*) or the entry is removed from the queue (indicated by the signal *rfq*).
- *write*: This bit is active for store instructions.
- *req*: This bit indicates that a read request to the main memory has to be made. It is initialized with one for misses and zero for hits or same-line misses (see equation (5.5) on page 121). The register is reset when the request is sent to the main memory (indicated by the signal *sent*).
- *rdy*: This bit is active if all bytes are valid and initialized with one for hits (see equation (5.5) on page 121). The bit is set if the memory returns the requested line (indicated by the signal *upd*).
- *con*: This field contains all information about the instruction which are not altered by the update queue. These are the address of the requested line *addr*, the tag of the instruction *tag*, and the opcode of the instruction.
- *Dpf*: This signal is active if the access caused a page fault interrupt. It is computed by the main memory and updated when the memory returns the requested line.

The circuit Byte_k for each byte k is shown in figure 5.14. For every byte the circuit saves a valid bit *bv*. This valid bit is initialized with the value of the signal $HC.UpdQ.bv_k$ (see page 121).

The control signals *fill*, *sent*, and *rfq* are computed by the update queue control circuit. The entry computes the control signals *lsl*, *upd*, and *allRet*. The signal *lsl* is active if the instruction in the entry accesses the same line as the instruction which enters the hit computation. It is computed by the static hit computation. For the computation of the signal *lsl* the addresses of the two instructions are compared and the full bit of the entry is checked. The signal *upd* has to be active if the result bus returns the data needed to update the entry. This is the case if the cache-line addresses of the result bus and the entry match and the full bit of the result bus is active. The signal *allRet* indicates, that all instructions preceding to the instruction in the entry have retired. This signal can be computed analogously to the signal *allRet* computed by the ROB control (see section 4.6.5).

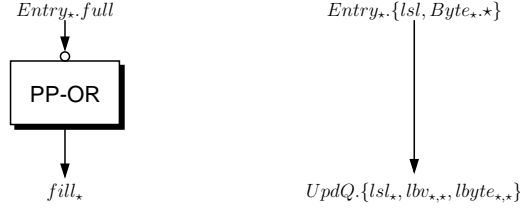


Figure 5.15: Update queue control (part 1)

Let $S_{DC} = 2^{s_{DC}}$ be the number of bytes of a cache line and $L_{ROB} = 2^{l_{ROB}}$ be the number ROB entries. The cost of an update queue entry is (without pipelining):

$$\begin{aligned}
 C(\text{UpdQ-Entry}) \leq & S_{DC} \cdot (C_{OR} + 10 \cdot C_{MUX} + 9 \cdot C_{REG}) \\
 & + 2 \cdot C(\text{EQ}(33)) + C(\text{EQ}(l_{ROB})) + C_{AND} \\
 & + C_{OR} + 3 \cdot C_{AND} + 5 \cdot C_{MUX} + (38 + l_{ROB}) \cdot C_{REG}.
 \end{aligned}$$

5.5.2 Control

The control circuit **UpdQ-Control** controls the entries of the update queue and computes the global outputs. The entries are controlled by the signals $fill_{*}$, $sent_{*}$, and rfq_{*} which are computed for every entry. The global outputs are the stall signal to the hit computation $stallOut$, the bus $UpdQ.MM_{*}$ which starts new read or write requests to the main memory, the bus $UpdQ.HC_{*}$ containing the local hit signals for the hit computation, the bus $UpdQ.Core_{*}$ which is used to update the cache core, and the bus $UpdQ.Sh4L_{*}$ which returns the results of a store instruction to the memory unit.

Figure 5.15 shows the first part of the update queue control. The bus $UpdQ.HC_{*}$ sent to the hit computation can be computed directly by renaming the outputs of the entries according to the naming convention used in section 5.3. The signals $fill_{*}$ controlling the movement of the instructions in the queue can be computed as in the reservation station. An instruction proceeds to the next entry whenever the next entry is empty. This is computed by the parallel-prefix OR in figure 5.15. The signal $stallOut$ is active if the first entry cannot be filled:

$$stallOut = \overline{fill_0}.$$

The remaining signals computed by the update queue control depend on which actions need to be performed for the instructions in the entries. If an instruction is not ready it must start a read request to the main memory to load the needed line. If a store instruction is ready it must update the main memory and the cache core. As soon as a store instruction has updated main memory and cache core, it can be removed from the update queue and be returned to the memory unit. Load instructions update the cache core as soon as they are ready. They can be removed from the update queue afterwards without being returned to the memory unit, since load misses are returned to the memory unit by the read queue.

In order to simplify the coherency, all actions except the read request to the main memory are done in one cycle. Thus, store instructions must update the main memory and the cache core, be returned to the memory unit and be removed from the queue at

the same time. Load misses are removed from the queue when they update the cache core.

If an instruction accesses a non-valid page, the main memory activates the page fault interrupt $MM.Dpf$ and returns invalid data. Yet the ready bit for the instructions is set in order to return the instructions to the memory unit. To stay consistent, all updates to the main memory or the cache core must be prevented if an instruction has caused a page fault. Note that a page fault can only occur on a read request. When the main memory is updated by a store instruction, the cache-line is already in the cache and therefore the access cannot cause a page fault.

The main memory is assumed to be single ported. Thus, at most one read or write request can be started in one cycle. The signal $MM.ack$ indicates that the main memory can accept a new request. If the instruction in entry i sends a read request to the main memory the signal $sent_i$ is activated. This signal resets the request bit of the instruction to prevent the same request to be sent twice. If a store in entry i updates the main memory the signal rfq_i is activated which removes the store from the queue.

If two entries need to start accesses of the same type, the entry with the higher index (i.e., which holds the older instruction) is preferred. Read accesses have a higher priority than write accesses. This is done to minimize the delay of loads, which are assumed to be performance critical.

To compute the control signals $sent_*$ and rfq_* for all entries the requests made by the update queue entries must be computed. Three different types of requests are possible. If an entry contains an access that is not ready (i.e., not all data of the cache-line is valid) it must start a read request. If an entry contains a ready store that may update the main memory (since all preceding instructions have retired) a write request must be started. A write request also indicates that the store needs to update the cache core, removed from the update queue and returned to the memory unit. If an entry contains a ready load it must start a update request in order to update the cache core and be removed from the update queue. Note that at any time only one of the three requests may be active.

The entry i needs to start a read request if it contains a valid access and the request bit req of the entry is set:

$$rreq_i = Entry_i.full \wedge Entry_i.req. \quad (5.7)$$

Note that the request bit is reset once the read request is granted in order to prevent that the access starts a second read request.

The entry i needs to start a write request to the main memory (indicated by $wreq_i$) if it is valid, a write instruction, all preceding instructions are retired, and all bytes are valid. Thus:

$$wreq_i = Entry_i.full \wedge Entry_i.write \wedge Entry_i.allRet \wedge Entry_i.rdy. \quad (5.8)$$

Since the signal $allRet$ can be active for at most one instruction, $wreq_i$ can be active for at most one entry i .

If the full bit of the entry i is active and the write bit is inactive the entry must start update request indicated by $creq_i$:

$$creq_i = Entry_i.full \wedge \overline{Entry_i.write} \wedge Entry_i.rdy.$$

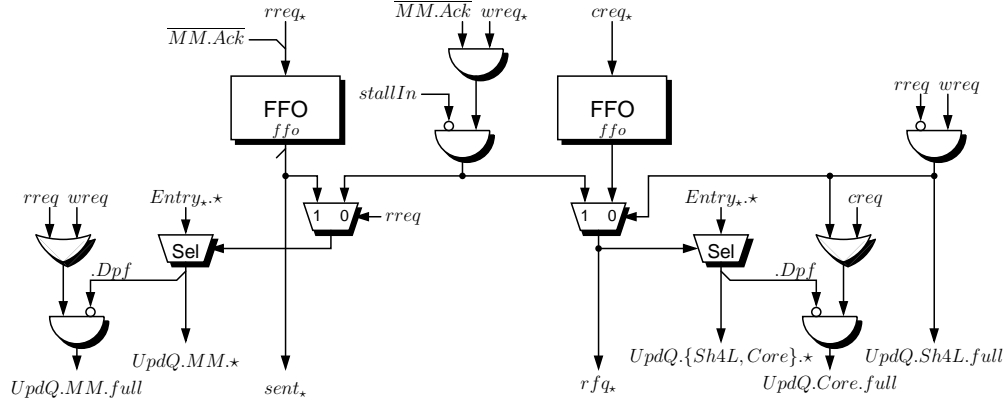


Figure 5.16: Update queue control (part 2)

From the construction of the update queue and the hit computation it follows that the bits req and rdy of a single entry cannot be active at the same time. Thus, at most one of the request signals may be active for an entry at a certain time.

A read request must be ignored if the main memory cannot accept a new request, i.e., if the signal $MM.Ack$ is inactive. A write request must be ignored if $MM.Ack$ is inactive or the store cannot be returned to the memory unit (indicated by $stallIn$). The signals $rreq_*$, $wreq_*$, and $creq_*$ are combined to overall request signals $rreq$, $wreq$, and $creq$ which are active if any of the entries wants to start a request of this type and the request does not have to be ignored:

$$rreq = \left(\bigvee_{i=0}^{e_{UQ}-1} rreq_i \right) \wedge MM.Ack, \quad (5.9)$$

$$wreq = \left(\bigvee_{i=0}^{e_{UQ}-1} wreq_i \right) \wedge MM.Ack \wedge \overline{stallIn}, \quad (5.10)$$

$$creq = \bigvee_{i=0}^{e_{UQ}-1} creq_i.$$

Read requests to the main memory have the highest priority. They are sent to the main memory even if an older write request wants to update the main memory. Note that this is consistent since the data of the write has already been forwarded in the hit computation. Updates to the cache core are forwarded to the hit computation and therefore not performance critical. Updates of the cache core by load instructions have lowest priority as the stalling the entry in the update queue does not prevent the read queue from returning the result to the memory unit.

Figure 5.16 shows the second part of the update queue control. The main function of this part of the control is to compute the busses $sent_*$ and rfq_* (in the center of the figure). For each request type the entry holding the oldest instruction with a request of this type is computed. For the signals $rreq_*$ and $creq_*$ this is done using a find-first-one circuit. Since at most one of the write requests $wreq_*$ may be active, the find-first-one circuit may be omitted for these signals.

The write requests are ignored if $MM.Ack$ is inactive or $stallIn$ is active. For the read requests $rreq_*$, the signal $\overline{MM.Ack}$ is used as most significant input of the find-first-one circuit computing the oldest instruction with a read request. Thus, if $MM.Ack$ is inactive only the most significant bit of the output of the find-first-one circuit is active. This bit is ignored and therefore no read request is sent if the signal $MM.Ack$ is inactive.

The read requests to the main memory have the highest priority. Therefore, the signals $sent_*$ can be computed directly from the outputs of the find-first-one circuit for the read requests. A store instruction can be removed from the update queue if a write request to the main memory can be done ($wreq$ is active) and no read request is done ($rreq$ is inactive). In this case the write requests are used to compute the signal rfq_* . Otherwise the outputs of the find-first-one circuit using the request signals $creq_*$ are used.

The control circuit in figure 5.16 also computes the busses to the main memory $UpdQ.MM.*$, the cache core $UpdQ.Core.*$, and the memory unit $UpdQ.Sh4L.*$. An access to the main memory is made (indicated by $UpdQ.MM.full$) if $rreq$ or $wreq$ are active and the selected instruction did not cause a page fault interrupt (indicated by the signal Dpf of the entry). Note that the signal Dpf cannot be active for read accesses since it can only be activated when the memory returns the result of the read request. If $rreq$ is active, the read requests are used to select the instruction which accesses the main memory, otherwise the write requests are used.

An instruction is returned to the memory unit (indicated by $UpdQ.Sh4L.full$ if a write request to the main memory is made. The cache core is updated (indicated by $UpdQ.Core.full$) if an instruction is returned to the memory unit or the queue contains a ready load instruction ($creq$ is active). The cache core must not be updated if the selected instruction caused a page fault. Cache core and memory unit are updated by the same instruction and therefore use the same data. The instruction is selected with the signals rfq_* .

5.5.3 Delay Optimizations

The computation of the signals $rreq$ and $wreq$ of an entry can be pipelined similar to the signal req in the reservation station entries (see section 4.2.3). The signals $rreq$ and $wreq$ must be reset if the control signals $sent$ respectively rfq are active or the update queue is cleared by the signal $clear$. Figure 5.17 shows the pipelined computation of the request signals $rreq$ and $wreq$ for an update queue entry. Note that the update queue is then pipelined in two dimensions.

In order to fit into one cycle, the delay of the signal rfq_* in figure 5.17 must be at most $\delta - D_{AND} - D_{MUX}$. Due to the pipelining the delay of the signals $wreq_*$ and $rreq_*$ can be assumed to be 0. The first stage of the circuit Sh4L is assumed to have a buffer circuit, thus the delay of the signal $stallIn$ is equal to D_{AND} . Using the proposed circuit for the update queue control bounds the stage depth δ to be at least (with the critical path going through the signal $wreq$ (see equation (5.10)) on the right

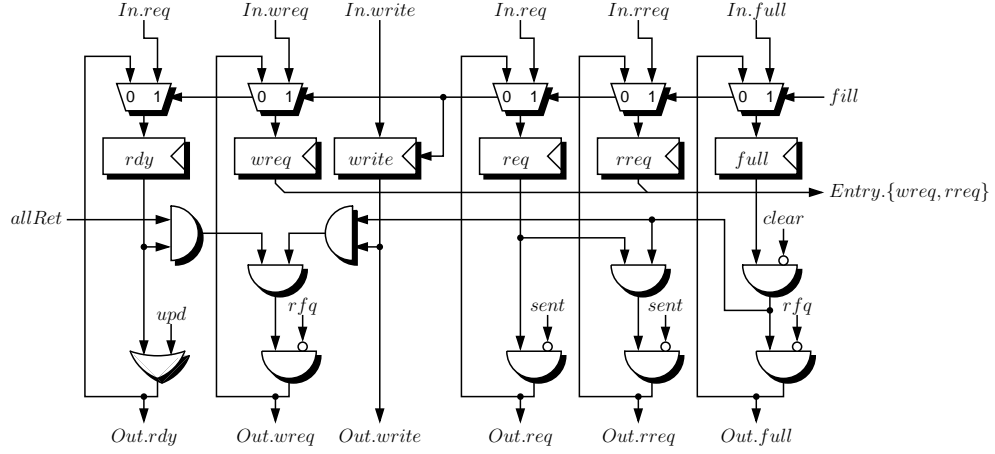


Figure 5.17: Pipelining of the request signals computation

side of figure 5.16):

$$\begin{aligned}
 \delta - D_{AND} - D_{MUX} &\geq D(rfq_*) \\
 &\geq D(wreq) + D_{AND} + D_{MUX} \\
 &\geq D(\text{OR-Tree}(e_{UQ})) + 2 \cdot D_{AND} + D_{MUX} \\
 \Leftrightarrow \quad \delta &\geq D(\text{OR-Tree}(e_{UQ})) + 3 \cdot D_{AND} + 2 \cdot D_{MUX}
 \end{aligned}$$

For $\delta = 5$ this does not hold true.

To reduce the delay of the critical path of the control circuit the following restriction to the update queue is introduced: only the last entry of the update queue may update the cache RAM or the main memory. Thus, no signal rfq_i has to be computed for $i < e_{UQ} - 1$. For load instructions this has no performance impact because the read queue can already return the instruction to the memory unit while the instruction still waits in the update queue for updating the cache core. However, it might happen that the update queue is full more often. Write instructions are only delayed if the write is the last active instruction and the data are valid before the instruction enters the last entry. For small stage depth and small numbers of entries, this is not assumed to have a significant performance impact.

Figure 5.18 shows the delay optimized version of the second part of the update queue control using the above restriction. The restriction allows the following modifications of the control circuit. If the last queue entry contains a ready load miss (indicated by $\overline{\text{Entry}_{e_{UQ}-1}.write} \wedge \text{Entry}_{e_{UQ}-1}.rdy$), it can update the cache core in any case. Thus the instruction can be removed from the queue. This can be done by overwriting the entry by activating the fill signal for this entry $fill_{e_{UQ}-1}$. The computation of the fill signals is adjusted accordingly. Thus, for the computation of the signal $rfq_{e_{UQ}-1}$ load misses do not have to be taken into account. Since only the last entry may update the main memory or the cache core, the page fault signal of the entry that does the update must not longer be computed with a select circuit. This reduces the delay of the full signals for main memory $UpdQ.MM.full$ and cache core $UpdQ.Core.full$.

Assume the delay of the request signals and the acknowledge signal $MM.ack$ is 0 and the delay of the stall signal $stallIn$ is D_{AND} . Then, the modifications reduce the

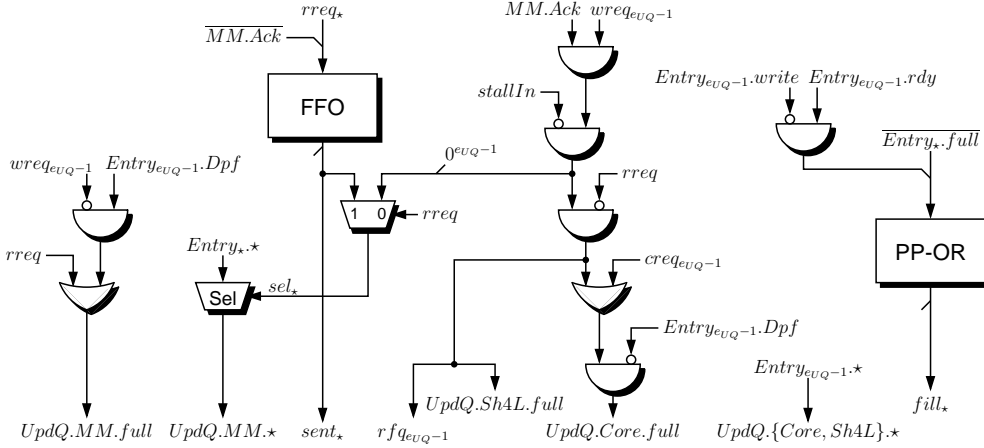


Figure 5.18: Delay optimized update queue control (part 2)

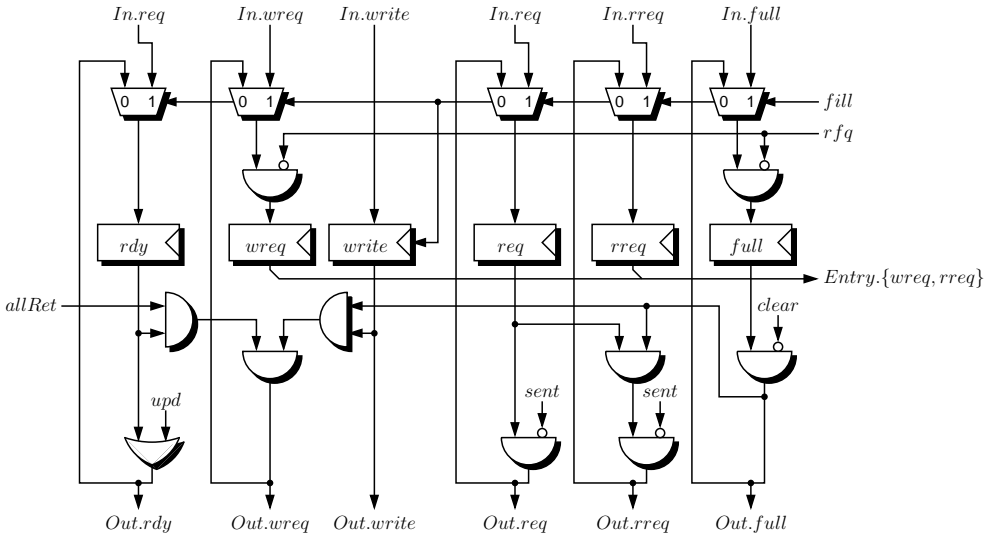


Figure 5.19: Optimized clearing of full bit and write request for the last entry

delay of the signal $rfq_{e_{UQ}-1}$ to:

$$\begin{aligned}
 D(rfq_{e_{UQ}-1}) &\leq \max\{D(rreq), 2 \cdot D_{AND}\} + D_{AND} \\
 &\stackrel{(5.9)}{\leq} \max\{D(\text{OR-Tree}(e_{UQ})) + D_{AND}, 2 \cdot D_{AND}\} + D_{AND} \\
 &\leq \max\{D(\text{OR-Tree}(e_{UQ})), D_{AND}\} + 2 \cdot D_{AND}
 \end{aligned}$$

If $rfq_{e_{UQ}-1}$ is active it follows that the last entry is not empty and does not contain a load miss. Hence, the signal $fill_{e_{UQ}-1}$ can not be active. This allows to move the AND-gates that clear the full and the write request signal of the last queue entry below the multiplexer controlled by $fill_{e_{UQ}-1}$. Figure 5.19 depicts the modification applied to figure 5.17. With this modification, the bound for δ regarding the signals rfq_* is reduced to:

$$\delta - D_{AND} \geq D(rfq_*) \quad (5.11)$$

$$\Leftrightarrow \delta \geq \max\{D(\text{OR-Tree}(e_{UQ})), D_{AND}\} + 3 \cdot D_{AND}. \quad (5.12)$$

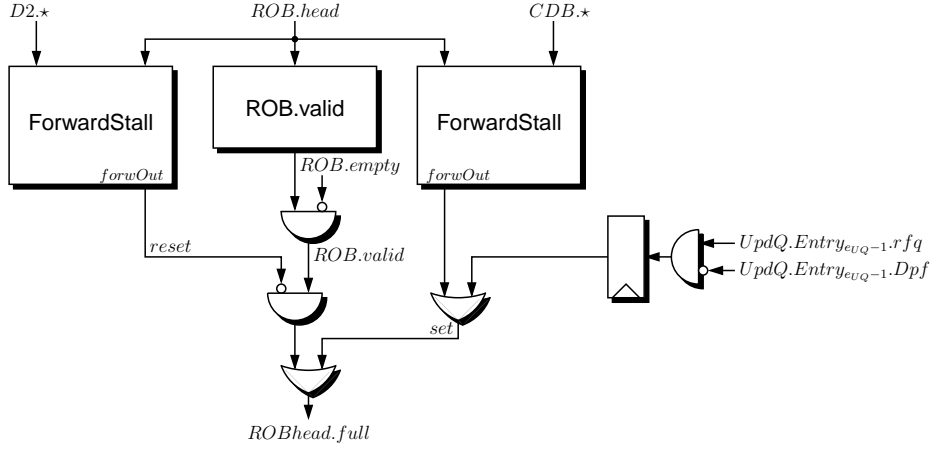


Figure 5.20: Optimized completion for stores

Note that the signals $fill_*$ do not depend on the signal $rfq_{e_{UQ}-1}$ in order to reduce the delay. Thus, if a store updates the cache core and the main memory the last update queue entry is not filled in the next cycle. Hence, store instructions can only complete every other cycle. This is not assumed to have a performance impact as store instructions must wait until they are the oldest active instruction anyway.

5.5.4 Optimized Completion for Store Instructions

If a store instruction does not cause an interrupt, it does not produce a result which would be forwarded to the succeeding instruction. Thus, it is not necessary to send a store instruction via the CDB. At the time the store instruction updates the cache core and the main memory, it must be at the head of the ROB. Thus, to complete a store instruction that did not cause an interrupt, it suffices to activate the signal $ROBhead.valid$ indicating that the instruction at the ROB head has completed.

The modified completion of store instructions has the following advantages. The number of cycles needed to complete stores can be reduced. If the instructions following the store have completed earlier, this can reduce the number of instructions in the ROB. Also store instructions do not block the CDB for instructions of which the result is needed by later instructions. The modifications to the ROB environment is shown in figure 5.20.

The simplest way to modify the update queue control accordingly is to set the full signal to the memory unit $UpdQ.Sh4L.full$ to 0 for store instructions if their page fault signal $Entry_{e_{UQ}-1}.Dpf$ is inactive. This modification does not affect the delay of the update queue. However, stores are then stalled by the input signal $stallIn$ even if they do not use the circuit $Sh4L$.

In order to complete store instructions even if the stall input is active, the computation of the signals $rfq_{e_{UQ}-1}$, $UpdQ.Sh4L.full$, and $UpdQ.Core.full$ must be adopted as shown in figure 5.21. Note that this modification increases the delay from the stall input to the signal $rfq_{e_{UQ}-1}$. Yet, if $stallIn$ directly comes out of a buffer circuit the delay of $rfq_{e_{UQ}-1}$ based on $stallIn$ is $D_{OR} + 3 \cdot D_{AND}$ and thus it holds $D(rfq_{e_{UQ}-1}) \leq \delta - D_{AND}$ for $\delta \geq 5$. Therefore, this second option can be used.

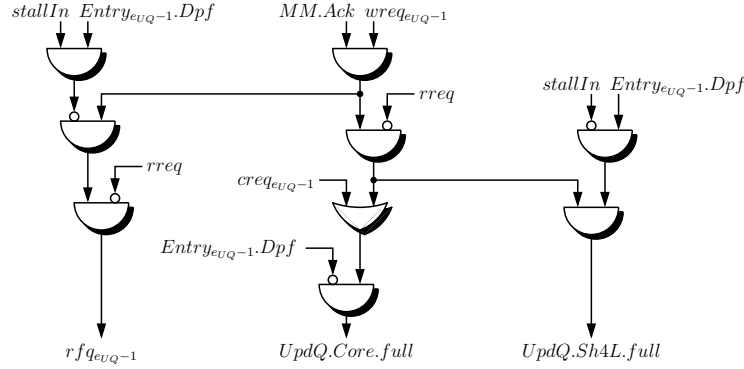


Figure 5.21: Update queue control with optimized completion

5.5.5 Cost and Delay

For the bound to δ given by the update queue, the stall input $stallIn$ is assumed to come directly out of a buffer circuit, i.e., the delay of $stallIn$ is assumed to be D_{AND} . Also it is assumed that a register is inserted after the computation of the signals $wreq_{e_{UQ}-1}$ and $rreq_*$. The delay of the signal $rfq_{e_{UQ}-1}$ may be at most $\delta - D_{AND}$ (see equation (5.11)). The modified computation of the signal $rfq_{e_{UQ}-1}$ from figure 5.21 increases the bound for δ to

$$\delta \geq \max\{D(\text{OR-Tree}(e_{UQ})), 2 \cdot D_{AND}\} + 3 \cdot D_{AND}. \quad (5.13)$$

The delay of the signals $sent_*$ (computed in figure 5.18) may be not larger than $\delta - D_{MUX} - D_{AND}$ (see figure 5.13). Thus, it must also hold:

$$\delta \geq D(\text{FFO}(e_{UQ} + 1)) + D_{AND} + D_{MUX}. \quad (5.14)$$

Note that the equations (5.13) and (5.14) imply a bound for the number of update queue entries e_{UQ} for a given δ .

Let the boolean variable p_{rrq} and p_{wrq} be zero if the requirements for the signals $rfq_{e_{UQ}-1}$ and $sent_*$ also hold if no register is added after the computation of the signals $rreq_*$ and $wreq_{e_{UQ}-1}$. If no registers are added the delay from the full bits of the entries through the signals $rreq_*$ and $wreq_{e_{UQ}-1}$ to the signal $rfq_{e_{UQ}-1}$ (which updates the full bit of entry $e_{UQ} - 1$) is (see equations (5.7), (5.8), and (5.9) and figure 5.21):

$$\begin{aligned} D(\text{Entry}_*.full \rightsquigarrow rreq_* \rightsquigarrow rfq_{e_{UQ}-1}) &\leq D_{AND} && (rreq_*) \\ &+ D(\text{OR-Tree}(e_{UQ})) \\ &+ D_{AND} && (rreq) \\ &+ D_{AND}, && (rfq_*) \\ D(\text{Entry}_*.full \rightsquigarrow wreq_{e_{UQ}-1} \rightsquigarrow rfq_{e_{UQ}-1}) &\leq 2 \cdot D_{AND} && (wreq_{e_{UQ}-1}) \\ &+ 3 \cdot D_{AND}. \end{aligned}$$

The delay of the path from the request bits of the entries through the signals $rreq_*$ to the signals $sent_*$ is (see equation (5.7) and figure 5.18):

$$\begin{aligned} D(\text{Entry}_*.req \rightsquigarrow rreq_* \rightsquigarrow sent_*) &\leq D_{AND} && (rreq_*) \\ &+ D(\text{FFO}(e_{UQ}+1)). && (sent_*) \end{aligned}$$

Thus, the variables p_{rrq} and p_{wrq} can be computed as:

$$p_{rrq} = \begin{cases} 0 & \text{if } \delta \geq \max\{D(\text{OR-Tree}(e_{UQ})) + 4 \cdot D_{AND}, \\ & D(\text{FFO}(e_{UQ} + 1)) + 2 \cdot D_{AND} + D_{MUX}\} \\ 1 & \text{else} \end{cases} ,$$

$$p_{wrq} = \begin{cases} 0 & \text{if } \delta \geq 6 \cdot D_{AND} \\ 1 & \text{else} \end{cases} .$$

The read request signal rrq_\star (see equation (5.7)) of an entry that is filled into the update queue can already be computed during the hit computation (for an additional cost of C_{AND}). Thus, it can be assumed that an access that is filled into the update queue can start a read request to the main memory within the following cycle.

Let c_{M2W} be the minimum number of cycles needed between the update of the update queue by the result bus of the main memory and the retiring of a store instruction. The number of cycles is defined by the delay of the path from the result bus to the full bit of the last entry of the update queue. The stall input $stallIn$ is not assumed to be timing critical, otherwise a buffer circuit can be inserted in the first stage of the memory unit. The delay of the path via the signal $wreq_{e_{UQ}-1}$ is (see equation (5.8) and figures 5.13 and 5.18) :

$$\begin{aligned} D(MM.\star \rightsquigarrow wreq_{e_{UQ}-1} \rightsquigarrow Entry_{e_{UQ}-1}.full') &\leq D(\text{EQ}(33)) && (Entry_\star.upd) \\ &+ D_{OR} && (Entry_\star.rdy) \\ &+ (2 + p_{wrq}) \cdot D_{AND} && (wreq_{e_{UQ}-1}) \\ &+ 3 \cdot D_{AND} && (rfq_{e_{UQ}-1}) \\ &+ D_{AND}. \end{aligned}$$

Thus:

$$c_{M2W} \leq \left\lceil \frac{D(MM.\star \rightsquigarrow wreq_{e_{UQ}-1} \rightsquigarrow Entry_{e_{UQ}-1}.full') - D_{MUX}}{\delta - D_{MUX}} \right\rceil .$$

For read instructions the delay of the path from the memory unit to the full bit of the last entry via the signal $fill_{e_{UQ}-1}$ is (see figures 5.13 and 5.21):

$$\begin{aligned} D(MM.\star \rightsquigarrow creq_{e_{UQ}-1} \rightsquigarrow Entry_{e_{UQ}-1}.full') &\leq D(\text{EQ}(33)) && (Entry_\star.upd) \\ &+ D_{OR} && (Entry_\star.rdy) \\ &+ D_{AND} + D_{OR} && (fill_{e_{UQ}-1}) \\ &+ D_{MUX} + D_{AND}. \end{aligned}$$

Let c_{M2R} be the minimum number of cycles between the requested data being in the last stage of the result bus and the update of the cache core by a load instruction. Then it holds:

$$c_{M2R} \leq \left\lceil \frac{D(MM.\star \rightsquigarrow creq_{e_{UQ}-1} \rightsquigarrow Entry_{e_{UQ}-1}.full') - D_{MUX}}{\delta - D_{MUX}} \right\rceil .$$

The delay of the outputs $UpdQ.MM.\star$ which does not fit into the cycle in which the entries are updated is added to the delay of the main memory. It depends on the signals sel_\star which select the entry sent to the main memory. The delay of the signals sel_\star depends on whether the read or write request is critical and whether registers are added after the request signals:

$$D(sel_\star) \leq \max\{(1 - p_{wrq}) \cdot 2 \cdot D_{AND} + 2 \cdot D_{AND}, \\ D(EQ(33)) + D_{OR} + 4 \cdot D_{AND} - (c_{M2W} - 1) \cdot \delta, \\ (1 - p_{rrq}) \cdot D_{AND} + D(OR-Tree(e_{UQ})) + D_{AND}, \\ D(EQ(33)) + D_{OR} + 2 \cdot D_{AND} + D(OR-Tree(e_{UQ})) \\ - (c_{M2R} - 1) \cdot \delta\} + D_{MUX}.$$

Then the following delay has to be added to the delay of the main memory:

$$D(MM)^+ \leq D(Sel(e_{UQ})) + D(sel_\star) - \delta.$$

The delay of the signal $stallOut$ sent to the hit computation is:

$$D(stallOut) \leq D(PP-OR(e_{UQ} + 2)). \quad (5.15)$$

If the optimized completion for stores is used and the control is modified as shown in figure 5.21 the delay of the stall input $stallIn$ is bounded by

$$\delta \geq D(stallIn) + 4 \cdot D_{AND}. \quad (5.16)$$

Note that if the stall input is computed with an AND-tree the first AND-gate can be merged into this tree. If this bound does not hold, the update queue control can not be modified as in figure 5.21 and thus stores are stalled by the signal $stallIn$ even if they are not returned to the memory unit (see section 5.5.4).

Let $S_{DC} = 2^{s_{DC}}$ be the number of bytes of a cache-line. The number of inputs of the update queue from the main memory are $33 + 8 \cdot S_{DC}$. The number of outputs to the update queue are $5 + 8 \cdot S_{DC}$. The additional cost for the update queue entries due to pipelining are approximated by:

$$C(UpdQ-Entry)^+ \leq (c_{M2W} - 1) \cdot (19 + 8 \cdot S_{DC}) \cdot (C_{MUX} + C_{REG}) + 2 \cdot C_{AND}.$$

The cost of the update queue control are:

$$C(UpdQ-Control) \leq C(FFO(e_{UQ} + 1)) + C(PP-OR(e_{UQ} + 1)) \\ + 2 \cdot C_{OR} + 6 \cdot C_{AND} \\ + (34 + 8 \cdot S_{DC}) \cdot C(Sel(e_{UQ})) \\ + e_{UQ} \cdot 3 \cdot C_{AND}.$$

The overall cost for the update queue are:

$$C(UpdQ) \leq e_{UQ} \cdot C(UpdQ-Entry) + C(UpdQ-Control).$$

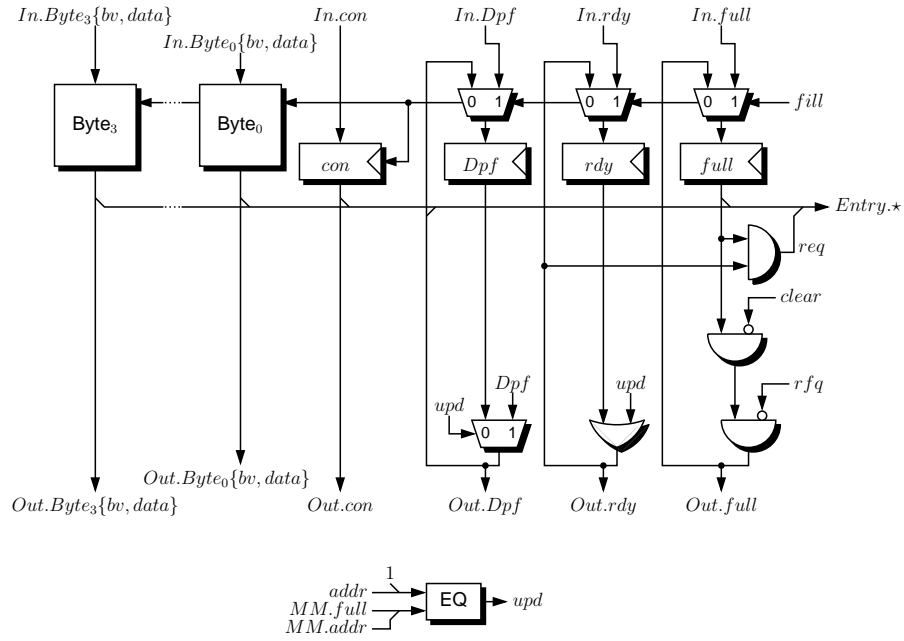


Figure 5.22: Read queue entry

5.6 Read Queue

The read queue holds one entry for each load miss. The load misses wait in the read queue until the needed line is obtained from the main memory. When the main memory sends the required cache-line, the entry is updated and the load can be returned to the circuit Sh4L of the memory unit (see figure 5.1). The main memory access will be started from the corresponding entry of the update queue. Note that due to same-line miss multiple loads and up to one store can share the same entry in the update queue (see section 5.3.5).

Similar to the update queue, the read queue entries have a valid bit for every byte of the requested word. The bytes that are not needed by the load access and the bytes that are valid at the time the load is filled into the read queue due to forwarding are marked valid already during the hit computation (see figure 5.8). In order to validate the other bytes, the read queue snoops on the result bus of the main memory.

If the result bus is delayed, the read queue must snoop on the last stage of the delayed result result bus. Otherwise it could happen that a load which is filled into the read queue misses the cache-line required by this load. This would be the case if the required line was requested by a preceding access and is already in the stages of the delayed result bus at the time the second load is sent to the read queue. Therefore, the read queue must snoop on the last stage of the delayed result bus. In order to improve the performance the read queue could additionally snoop on the un-delayed result bus. This optimization is not discussed in detail.

The general design of the read queue is very similar to the update queue or the reservation stations. Figure 5.22 shows a read queue entry. The read queue has one **Byte** sub-circuit for each of the four bytes of a word. The registers *con*, *Dpf*, *rdy* and *full* have the same meaning as the corresponding registers of the update queue.

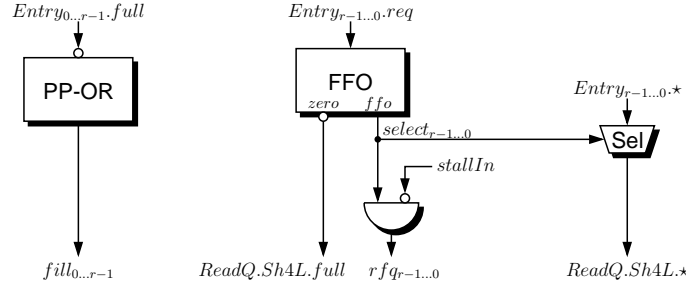


Figure 5.23: Read queue control

If the signals $full$ and $Out.rdy$ are active, the load in the entry can be returned to the memory unit. This is indicated by the signal req .

The control of the read queue (see figure 5.23) is built analogously to the control of the reservation stations (see section 4.2.2). An entry can be filled if it is not full. A find-first-one circuit selects the oldest load (i.e., the load in the entry with highest index) for which the request bit req is active. If the read queue is not stalled, the load can be returned to the circuit $Sh4L$ of the memory unit. Therefore, the output $select_*$ of the find-first-one circuit is AND-ed with the negated stall input to compute the signals rfq_* which clears the entry containing the load. The signals $select_*$ can be used to compute the output $ReadQ.Sh4L.*$ of the read queue, which is sent to the memory unit. The full bit of the bus to the memory unit is set if at least one entry is ready (indicated by the negated zero output of the find-first-one circuit).

If the number of entries of the read queue e_{RQ} is one, the same simplifications as for the reservation station can be made to reduce the delay requirements for the signals rfq_* .

5.6.1 Cost and Delay

Let e_{RQ} be the number of read queue entries. The computation of the request signal $req_* := full_* \wedge rdy_*$ can be pipelined as in the reservation station (see section 4.2.3). Then, the maximum number of entries of the read queue e_{RQ} is bounded by:

$$\delta \geq \begin{cases} 3 \cdot D_{AND} & \text{if } e_{RQ} = 1 \\ \max\{D_{AND}, D(\text{FFO}(e_{RQ}))\} + 2 \cdot D_{AND} + D_{MUX} & \text{if } e_{RQ} > 1 \end{cases} \quad (5.17)$$

The delay of the stall output $stallOut$ is:

$$D(stallOut) \leq D(\text{PP-OR}(e_{RQ})). \quad (5.18)$$

The variable p_{rq} is 1, if a register needs to be added in the computation of the signal req_* , otherwise 0. Thus:

$$p_{rq} = \begin{cases} 0 & \text{if } \delta \geq D(\text{FFO}(e_{RQ})) + 3 \cdot D_{AND} + D_{MUX} \vee e_{RQ} = 1 \\ 1 & \text{else} \end{cases}.$$

Let c_{M2R} be the minimum number of cycles between the last stage of the result bus containing the needed data and a read miss being returned to the memory unit.

This number depends on the delay of the path from the result bus $MM.\star$ to the full bits of the entries. The delay of this path for $e_{RS} > 1$ is:

$$\begin{aligned} D(MM.\star \rightsquigarrow Entry_{\star}.full') &\leq D(\mathbf{EQ}(33)) && (Entry_{\star}.upd) \\ &+ D_{OR} && (Entry_{\star}.rdy) \\ &+ (1 + p_{rq}) \cdot D_{AND} && (req_{\star}) \\ &+ D(\mathbf{FFO}(e_{RQ})) + D_{AND} && (rfq_{\star}) \\ &+ D_{AND} + D_{MUX}. \end{aligned}$$

If $e_{RS} = 1$ the delay of the path is:

$$\begin{aligned} D(MM.\star \rightsquigarrow Entry_{\star}.full') &\leq D(\mathbf{EQ}(33)) && (Entry_{\star}.upd) \\ &+ D_{OR} && (Entry_{\star}.rdy) \\ &+ D_{AND} && (req_{\star}) \\ &+ D_{AND} && (rfq_{\star}) \\ &+ D_{AND}. \end{aligned}$$

It holds:

$$c_{M2Q} \leq \left\lceil \frac{D(MM.\star \rightsquigarrow Entry_{\star}.full') - D_{MUX}}{\delta - D_{MUX}} \right\rceil.$$

The delay of the outputs $ReadQ.Sh4L.\star$ to the memory unit which does not fit into the cycle where the entries are updated is added to the delay of the circuit **Sh4L**. The additional delay depends on the delay of the signals rfq_{\star} . The delay of rfq_{\star} is:

$$\begin{aligned} D(rfq_{\star}) &\leq \max\{\max D(stallIn), (1 - p_{rq}) \cdot D_{AND} + D(\mathbf{FFO}(e_{RQ}))\}, \\ &D(MM.\star \rightsquigarrow Entry_{\star}.full') - (c_{M2Q} - 1) \cdot (\delta - D_{MUX}). \end{aligned}$$

Then the additional delay for the circuit **Sh4L** is:

$$D(\mathbf{Sh4L})^+ \leq \max\{0, D(rfq_{\star}) + D(\mathbf{Sel}(e_{RQ})) - \delta\}.$$

The delay of the stall input $stallIn$ from the memory unit is bounded by:

$$\delta \geq D(stallIn) + \begin{cases} 2 \cdot D_{AND} & \text{if } e_{RS} = 1 \\ 2 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS} > 1 \end{cases}. \quad (5.19)$$

The number of inputs of a read queue entry from the main memory are $33 + 8 \cdot L$. The number of outputs to the read queue control are 35. The cost of a read queue entry are approximated by:

$$\begin{aligned} C(\mathbf{ReadQ-Entry}) &\leq 4 \cdot (C_{OR} + 10 \cdot C_{MUX} + 9 \cdot C_{REG}) + C(\mathbf{EQ}(33)) + C_{OR} \\ &+ 2 \cdot C_{AND} + 4 \cdot C_{MUX} + (36 + l_{ROB}) \cdot C_{REG} \\ &+ (c_{M2R} - 1) \cdot (34 + 4 \cdot L) \cdot (C_{MUX} + C_{REG}). \end{aligned}$$

The cost of the read queue control is:

$$\begin{aligned} C(\mathbf{ReadQ-Control}) &\leq C(\mathbf{FFO}(e_{RQ})) + C(\mathbf{PP-OR}(e_{RQ})) \\ &+ 2 \cdot e_{RQ} \cdot D_{AND} + (67 + l_{ROB}) \cdot C(\mathbf{Sel}(e_{RQ})). \end{aligned}$$

The overall cost for the read queue is:

$$C(\mathbf{ReadQ}) \leq e_{RQ} \cdot C(\mathbf{ReadQ-Entry}) + C(\mathbf{ReadQ-Control}).$$

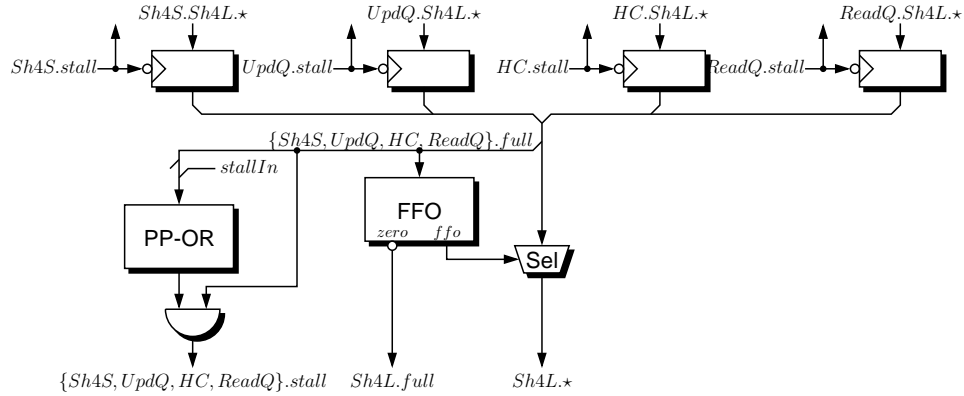


Figure 5.24: Arbiter circuit to the memory unit

5.7 Stall Computation

Data can be returned from the data cache to the memory unit by the hit computation, the read queue, and the update queue. Additionally instructions are sent directly from the circuit *Sh4S* to the output of the data cache in case of a misaligned access. A small arbiter circuit is needed to select between the busses from the four circuits. Loads are assumed to be most performance critical. Therefore the read queue has the highest priority, followed by the hit computation and the update queue. Misaligned accesses have the lowest priority as they cause an interrupt anyway.

The arbiter circuit can be seen as part the circuit *Sh4L* and is added to its cost and delay. The circuit is depicted in figure 5.24. The find-first-one circuit selects the input with the highest priority that wants to send an access (indicated by the full bit). An input is stalled if the full bit of the input is active and the full bit of any input with higher priority (computed by a parallel-prefix OR) or the stall input of the circuit *Sh4L* is active.

The total delay of the circuit *Sh4L* (including the additional delay from the queues) is

$$D(Sh4L) \leq \max\{D(\text{FFO}(4)), D(\text{Sel}(e_{RQ})) + D(ReadQ.rfq_{\star}) - \delta\} + D(\text{Sel}(4)) + 4 \cdot D_{MUX}.$$

The input stall signal of the hit computation *HC* is defined as

$$\begin{aligned} HC.stallIn &= (Sh4L.full \wedge Sh4L.HC.stall) \\ &\vee (ReadQ.full \wedge ReadQ.stallOut) \\ &\vee (UpdQ.full \wedge UpdQ.stallOut). \end{aligned}$$

The input stall signal for the circuit *Sh4S* is defined as

$$Sh4S.stallIn = (Dmal \wedge Sh4L.Sh4S.stall) \vee (\overline{Dmal} \wedge HC.stallOut). \quad (5.20)$$

5.8 Cost and Delay

Up to one buffer circuit is inserted into each of circuits *Sh4S*, *Sh4L*, and *HC*. Let the variables b_{Sh4S} , b_{Sh4L} , and b_{HC} be one if a buffer circuit is inserted in the respective

circuit, otherwise the variables are zero. Then the number of stages c_{Sh4L} , c_{Sh4S} , and c_{HC} for the respective circuits are

$$\begin{aligned} c_{Sh4L}(b_{Sh4L}) &= \lceil (D(Sh4L) + b_{Sh4L} \cdot D_{MUX}) / \delta \rceil, \\ c_{Sh4S}(b_{Sh4S}) &= \lceil (D(Sh4S) + b_{Sh4S} \cdot D_{MUX}) / \delta \rceil, \\ c_{HC}(b_{HC}) &= \lceil (D(HC) + b_{HC} \cdot D_{MUX}) / \delta \rceil. \end{aligned}$$

Let n be the number of functional units and t_L be the number of inputs of the first stage of the arbiter tree in the completion circuit (see equation (4.21) on page 68). Then the delay of the input stall signal $Mem.stallIn$ from the completion phase to the functional unit is (see equation (4.22) on page 70)

$$D(Mem.stallIn) \leq \begin{cases} D_{AND} + D(\text{FLO}(\min\{t_L, n\})) + D_{MUX} & \text{if } t_L > 2 \\ D_{AND} + D_{OR} & \text{if } t_L = 2 \end{cases}.$$

Then for the four stall outputs of the circuit **Sh4L** holds:

$$\begin{aligned} D(Sh4L.ReadQ.stall(b_{Sh4L})) &\leq \max\{D(Mem.stallIn), \\ &\quad D(\text{AND-Tree}(c_{Sh4L}(b_{Sh4L}) + 1))\} \\ &\quad + D_{AND}, \\ D(Sh4L.HC.stall(b_{Sh4L})) &\leq \max\{D(Mem.stallIn), \\ &\quad D(\text{AND-Tree}(c_{Sh4L}(b_{Sh4L}) + 1))\} \\ &\quad + D_{AND} + D_{OR}, \\ D(Sh4L.UpdQ.stall(b_{Sh4L})) &\leq \max\{D(Mem.stallIn), D_{OR} + D_{AND}, \\ &\quad D(\text{AND-Tree}(c_{Sh4L}(b_{Sh4L}) + 1))\} \\ &\quad + D_{AND} + D_{OR}, \\ D(Sh4L.Sh4S.stall(b_{Sh4L})) &\leq \max\{D(Mem.stallIn), 2 \cdot D_{OR} + D_{AND}, \\ &\quad D(\text{AND-Tree}(c_{Sh4L}(b_{Sh4L}) + 1))\} \\ &\quad + D_{AND} + D_{OR}. \end{aligned}$$

The delay of the stall output of the hit computation is (see equations (5.15) and (5.18) for the delay of the stall outputs of the queues)

$$\begin{aligned} D(HC.stallOut(b_{HC})) &\leq \max\{\max\{D(UpdQ.stallOut), D(ReadQ.stallOut)\} \\ &\quad + D_{AND} + 2 \cdot D_{OR}, \\ &\quad D(Sh4L.HC.stall) + D_{AND} + D_{OR}, \\ &\quad D(\text{AND-Tree}(c_{HC}(b_{HC}))), \} \\ &\quad + D_{AND}. \end{aligned}$$

Note that the delay of the path from the hit computation to the queues is only D_{MUX} . Thus, inserting a buffer circuit at the output registers of the circuit **HC** to the queues does not increase the delay. These output registers for cache misses can be stalled independently from the output registers for cache hits to the circuit **Sh4L** (which are

the input registers of the circuit **Sh4L**). Then the stall outputs of the queues must obey the following restriction:

$$\delta \geq \max\{D(\text{UpdQ.stallOut}), D(\text{ReadQ.stallOut}), D_{OR} + D_{AND}\} \\ + D_{AND} + D_{OR}$$

which holds true if the restrictions for the queues (equations (5.13) and (5.17)) are fulfilled. Thus, by inserting a buffer circuit at the output registers to the queues the delay of the stall output of the circuit **HC** can be reduced to:

$$D(\text{HC.stallOut}(b_{HC})) \leq \max\{2 \cdot D_{AND} + 2 \cdot D_{OR}, \\ D(\text{Sh4L.HC.stall}) + D_{AND} + D_{OR}, \\ D(\text{AND-Tree}(c_{HC}(b_{HC}))), \} \\ + D_{AND}.$$

Finally for the stall output of the circuit **Sh4S** respectively the memory unit it holds true:

$$D(\text{Sh4S.stallOut}(b_{Sh4S}, b_{HC})) \leq \max\{\max\{D(\text{HC.stallOut}(b_{HC})) \\ D(\text{Sh4L.Sh4S.stall})\} + D_{MUX}, \\ D(\text{AND-Tree}(c_{Sh4S}(b_{Sh4S})))\} + D_{AND}.$$

Let e_{RQ} be the number of entries of the read queue and e_{RS_1} be the number of entries of the memory reservation station. Due to the restrictions of read queue (equation (5.19)), update queue (equation (5.16)) and reservation station (equation (4.15) on page 62) regarding the delay of the stall inputs, without any buffer circuit the following equations must hold:

$$\delta \geq D(\text{Sh4L.ReadQ.stall}(0)) + \begin{cases} 2 \cdot D_{AND} & \text{if } e_{RQ} = 1 \\ 2 \cdot D_{AND} + D_{MUX} & \text{if } e_{RQ} > 1 \end{cases}, \quad (5.21)$$

$$\delta \geq D(\text{Sh4L.UpdQ.stall}(0)) + 3 \cdot D_{AND}, \quad (5.22)$$

$$\delta \geq D(\text{Sh4S.stallOut}(0, 0)) + \begin{cases} 2 \cdot D_{AND} & \text{if } e_{RS_1} = 1 \\ 3 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS_1} > 1 \end{cases}. \quad (5.23)$$

Note that the restriction for the update queue can be reduced as the first AND-gate on the stall path of the update queue can be merged into the computation of the signal Sh4L.UpdQ.stall using the distributive law.

The bounds for δ are successively reduced by setting first b_{Sh4L} , then b_{HC} , and finally b_{Sh4S} to one. The position of the buffer circuit in the circuit **Sh4L** depends on δ . For $\delta = 5$ the buffer circuit is placed in the first stage, for $\delta > 5$ it is placed in the second stage.

If $\delta = 5$ it must hold $t_L = 2$ (see equation (4.21) on page 68) and therefore $D(\text{Mem.stallIn}) = D_{AND} + D_{OR}$. Hence, the stall input from the completion phase is uncritical. For stall signal of the first stage of the circuit **Sh4L** it must hold true:

$$\delta \geq \max\{3 \cdot D_{OR}, D(\text{AND-Tree}(c_{Sh4L}(1))) + D_{AND}\} + D_{OR} + D_{AND}$$

which is assumed to hold for $\delta = 5$. The equations (5.21) and (5.22) then hold by construction of read queue and update queue. The stall output to the circuit HC and Sh4S come directly out of buffer circuits, thus:

$$D(\text{Sh4L}.\{\text{HC}, \text{Sh4S}\}.\text{stall}) = D_{AND}.$$

If $\delta > 5$ for the stall input from the completion phase $D(\text{Mem}.\text{stallIn})$ it can only be guaranteed that $D(\text{Mem}.\text{stallIn}) \leq \delta - D_{AND}$. Only the first stage of the circuit Sh4L can generate a stall. Thus, if the buffer circuit is placed in the second stage, the stall input fulfills the requirements for the stall signals of stage two and above (see section 2.5.4). The delay of the stall outputs of the circuit Sh4L then are:

$$\begin{aligned} D(\text{Sh4L}.\text{ReadQ}.\text{stall}) &= D_{AND} + D_{AND}, \\ D(\text{Sh4L}.\text{HC}.\text{stall}) &= D_{AND} + D_{OR} + D_{AND}, \\ D(\text{Sh4L}.\text{UpdQ}.\text{stall}) &= \max\{D_{AND}, D_{OR}\} + D_{OR} + D_{AND}, \\ D(\text{Sh4L}.\text{Sh4S}.\text{stall}) &= \max\{D_{AND}, 2 \cdot D_{OR}\} + D_{OR} + D_{AND}. \end{aligned}$$

Thus, the equations (5.21) and (5.22) hold for $\delta > 5$.

Buffer circuits are inserted into the circuits HC and Sh4S if the equation (5.23) does not hold for the new delay of the stall inputs to the circuits HC and Sh4S. First a buffer circuit is inserted into the first stage of the circuit HC. This replaces the equation (5.23) by:

$$\delta \geq \max\{2 \cdot D_{AND} + 2 \cdot D_{OR}, D(\text{AND-Tree}(c_{HC}(1) + 3))\} + D_{AND}, \quad (5.24)$$

$$\begin{aligned} \delta \geq \max\{\max\{D_{AND}, D(\text{Sh4L}.\text{Sh4S}.\text{stall})\} + D_{MUX}, \\ D(\text{AND-Tree}(c_{Sh4S}(0)))\} \\ + D_{AND} + \begin{cases} 2 \cdot D_{AND} & \text{if } e_{RS1} = 1 \\ 2 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS1} > 1 \end{cases}. \end{aligned} \quad (5.25)$$

Note that the number of cycles c_{HC} needed for the hit computation is mainly determined by the delay of the cache core (see equation 5.6). Yet, the equation (5.24) is assumed to hold true for all $\delta \geq 5$ for a reasonably large data cache. If the equation (5.25) does not hold, a buffer circuit is inserted into the first stage of the circuit Sh4S. For $\delta \geq 5$ the first AND-gate of equation (5.20) for the computation of the stall input of the circuit Sh4S can be merged into the computation of the signal $\text{Sh4L}.\text{Sh4S}.\text{stall}$. Then the equation (5.25) can be replaced by:

$$\begin{aligned} \delta \geq \max\left\{ \begin{cases} D(\text{Sh4L}.\text{Sh4S}.\text{stall}) + D_{MUX} & \text{if } \delta = 5 \\ D(\text{Sh4L}.\text{Sh4S}.\text{stall}) + D_{OR} & \text{if } \delta > 5 \end{cases} \right\}, \\ D(\text{AND-Tree}(c_{Sh4S}(1) + 2))\} + D_{AND}. \end{aligned} \quad (5.26)$$

which is assumed to be fulfilled for all $\delta \geq 5$. The requirements for the stall input of the reservation station then hold true by construction of the reservation station.

The inputs to the circuit Sh4S are the full bit, the tag, the operands, a 16 bit immediate constant and 8 bit control signals ($89 + l_{ROB}$ bits in total). The outputs are

the full bit, the tag, the write data, the effective address, the bus ub_* and 8 bit control signals ($77 + l_{ROB}$ bits in total). The cost of the circuit **Sh4S** including pipelining is approximated by (see appendix D.4.1):

$$\begin{aligned} C(\text{Sh4S}) \leq & C(\text{Add}(32)) + 2 \cdot C(\text{Dec}(2)) + C(\text{HDec}(2)) \\ & + 104 \cdot C_{MUX} + 11 \cdot C_{OR} + 10 \cdot C_{AND} \\ & + (c_{Sh4S}(b_S) - 1) \cdot (83 + l_{ROB}) \cdot C_{REG}. \end{aligned}$$

The inputs to the circuit **Sh4L** are the full bit, a tag, 32 data bits, and 8 control bits for from the read queue as well as the hit computation, and a full bit, a tag, an interrupt signal, and the effective address from **Sh4S** and update queue ($148 + 4 \cdot l_{ROB}$ bits in total). The outputs are the full bit, the tag, 64 bit data, and 2 interrupts signals ($67 + l_{ROB}$ bits in total). Thus, the total cost of the circuit **Sh4L** including pipelining is (see appendix D.4.2):

$$\begin{aligned} C(\text{Sh4L}) \leq & C(\text{Inc}(2)) + C(\text{Dec}(2)) + C(\text{Sel}(4)) \\ & + 186 \cdot C_{MUX} + 3 \cdot D_{OR} + D_{AND} \\ & + C(\text{PP-OR}(4)) + C(\text{FFO}(4)) + (34 + l_{ROB}) \cdot C(\text{Sel}(4)) \\ & + (c_{Sh4L}(b_L) - 1) \cdot \lceil (215 + 5 \cdot l_{ROB})/2 \rceil \cdot C_{REG}. \end{aligned}$$

Let S_{DC} be the width of the data cache-lines. The number of inputs of the hit computation is $77 + l_{ROB}$. The outputs are the full bit, the tag, effective address, the hit signal, the cache-line and byte-valid signals to the update queue and the word with valid signals to the read queue ($38 + l_{ROB} + 9 \cdot S_{DC}$ bits in total). Then cost of the data cache are:

$$\begin{aligned} C(\text{DCache}) \leq & D(\text{HC}) + D(\text{Core}) + D(\text{UpdQ}(e_{UQ})) + D(\text{ReadQ}(e_{EQ})) \\ & + c_{HC}(b_H) \cdot (105 + 2 \cdot l_{ROB} + 9 \cdot S_{DC}) \cdot C_{REG}. \end{aligned}$$

The overall cost of the memory unit is:

$$C(\text{Mem}) \leq D(\text{Sh4S}) + D(\text{DCache}) + D(\text{Sh4L}).$$

Chapter 6

Instruction Fetch

This chapter presents the circuits which perform the instruction fetch. The instruction fetch mechanism and the branch prediction used by the $DLX_{\pi+}$ is presented in section 6.1. Section 6.2 describes the instruction fetch unit, which delivers a parallel instruction stream. The instruction fetch queue presented in section 6.3 serializes the stream and sends the instructions to the instruction register environment described in section 6.4. The branch checking unit presented in section 6.5 checks whether branch predictions made by the instruction fetch unit are correct and initiates a rollback in case of a misprediction. The flush of the processor needed for the rollback is detailed in section 6.6.

6.1 Instruction Fetch Mechanism

6.1.1 Overview

The instruction fetch loads the instructions to be executed from the main memory. If a branch instruction (i.e., a conditional branch or a jump) is fetched, the address of the next instruction to be fetched is not known. Since waiting for the core to compute the target of the branch instruction would take several cycles, the address of the next instruction is predicted. To ensure the correctness, the branch checking unit (BCU) verifies the prediction and initiates a rollback in case it detects a misprediction.

The rollback of a mispredicted branch instruction is done in two steps. If the BCU detects a misprediction, the program counter (PC) pointing to the address of the next instruction is set to the correct branch target of the branch instruction. To prevent wrongly fetched instruction from initiating another rollback, all branch instruction following the mispredicted branch are invalidated. This is done by clearing the BCU, the reservation station of the BCU, and the decode stage. Since decode and dispatch of branch instructions is done in order no instruction preceding to the branch instruction is cleared.

The wrongly fetched instructions may have altered the producer tables. Thus, before new instructions may enter the decode phase, the producer tables must be restored. The restore can be done when the mispredicted branch instruction retires. Since retire is done in order, all remaining instructions in the core have been wrongly fetched after the mispredicted branch and thus have to be invalidated. Hence, no valid instructions

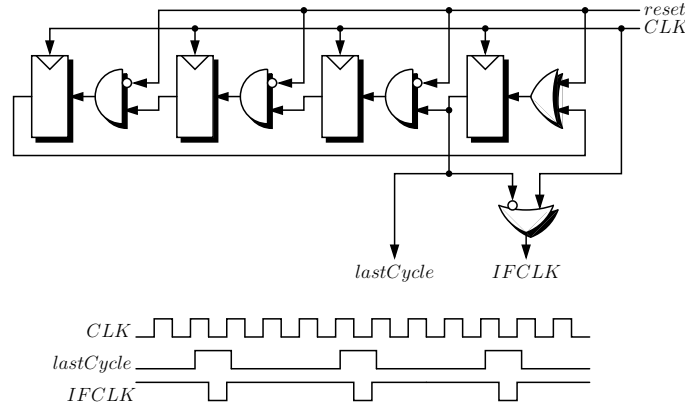


Figure 6.1: Generation of the slow clock for $m = 4$

are in the core after clearing and the producer table can simply be reset, i.e. all valid bits are set to one. As soon as the producer table is reset, new instructions fetched from the correct branch target address can enter the decode phase.

6.1.2 Clocking of the Instruction Fetch

To be able to fetch one instruction per cycle, the instruction fetch must predict the address of the next instruction within one cycle. If the stage depth δ goes down to 5 gate delays a nontrivial branch prediction is not possible. Therefore, the cycle time of the clock of the instruction fetch circuit ($IFCLK$) is an integer multiple m of the cycle time of the core clock (CLK). In each cycle of the slow clock $IFCLK$ the instruction fetch delivers up to $FS = 2^{fs}$ instructions, with $FS \geq m$. The parallel instruction stream is loaded into an instruction fetch queue (IFQ), from which every CLK -cycle one instruction can be sent to the decode phase.

The divided clock $IFCLK$ can be produced, e.g., by the circuit shown in figure 6.1. This circuit also provides the signal *lastCycle* which is active during the last cycle of the fast clock before the rising edge of the slow clock. Note that the circuit in figure 6.1 does not generate a symmetric clock as shown in the timing diagram. This is not problematic as edge-triggered registers are used.

6.1.3 Branch Prediction

The instruction fetch mechanism is based on a design of an instruction fetch with branch prediction for super-scalar processors proposed in [Yeh93]. As required in the previous section, this mechanism delivers multiple instructions per cycle. The branch prediction is based on the division of the instruction stream into *basic blocks*. A basic block is a sequence of non-branch instructions followed by a single branch instruction. Since only the last instruction of a basic block is a branch, the instructions of a basic block are stored in consecutive memory cells. Therefore, if the start address of the basic block is known, the whole basic block can be fetched without any branch prediction. The start address of the next basic block is predicted in parallel to the fetching of the basic block.

The instruction cache delivers up to one aligned block of size $FS = 2^{fs}$ words

respectively instructions per cycle, called *fetch block*. A basic block may be distributed over multiple fetch blocks. In this case the fetching of the basic block has to be done in multiple cycles. If the current fetch block does not contain a branch instruction, the instruction fetch continues at the start of the next fetch block.

The address used to fetch a fetch block is called *fetch address*. The branch prediction uses the fetch address to predict the fetch address for the next cycle and the number of instructions which belong to the current basic block. If the fetch block is the last fetch block of a basic block this number identifies the address of the branch instruction of the basic block.

It is also possible to identify the branches by the start address of their basic block. This scheme is called basic block addressing in contrast to the fetch address scheme. More detailed information on the basic block addressing can be found, e.g., in [Yeh93].

The prediction can be overruled by two events. If the branch checking unit detects a misprediction, the instruction fetch continues at the correct target of the mispredicted branch. The prediction circuit is then updated with the corrected result. If an interrupt occurs, the instruction fetch continues at the start of the interrupt service routine.

The prediction circuit presented in this thesis computes the prediction in one cycle. To implement more complex prediction circuits with a reasonable cycle time, the prediction must be done in multiple cycles. Then fetching usually continues by default at the beginning of the next fetch block. If the prediction circuit detects a taken conditional branch or a jump, the already fetched instructions are invalidated and fetching continues at the target of the branch instruction. Details on branch prediction schemes requiring multiple cycles are not treated in this thesis.

6.2 Instruction Fetch Unit

6.2.1 Overview

Figure 6.2 depicts an overview of the instruction fetch circuit. The circuit comprises the instruction cache *ICache* and the circuit *NextPC* which computes the PC for the next cycle using branch prediction. The circuit *NextPC* computes the fetch-PC fPC containing the current fetch address. The fetch-PC is used to access the cache. The outputs of the two circuits are sent to the instruction fetch queue (IFQ), which serializes the parallel instruction stream (see section 6.3).

The instruction cache delivers the $FS = 2^{fs}$ word wide fetch block, the page fault signal Ipf , and a hit signal. As long as the hit signal is zero the other signals are not valid and the instruction fetch has to be stalled.

The instruction fetch queue expects the valid instructions to be right aligned. Therefore, the data returned by the instruction cache is shifted to the right if the fetch address does not point to the beginning of an aligned fetch block. The first instruction in the fetch block which belongs to the current basic block is at word position $\langle fPC[fs + 1 : 2] \rangle$. Thus, by shifting the fetch block by $\langle fPC[fs + 1 : 2] \rangle$ words to the right, this first instruction is at the bit position $[31 : 0]$ of the shifter output. The output of the shifter is then divided into the 32 bit wide busses $instr_i$ for $i \in \{0, \dots, FS - 1\}$ where $instr_0$ contains the first instruction in the fetch block

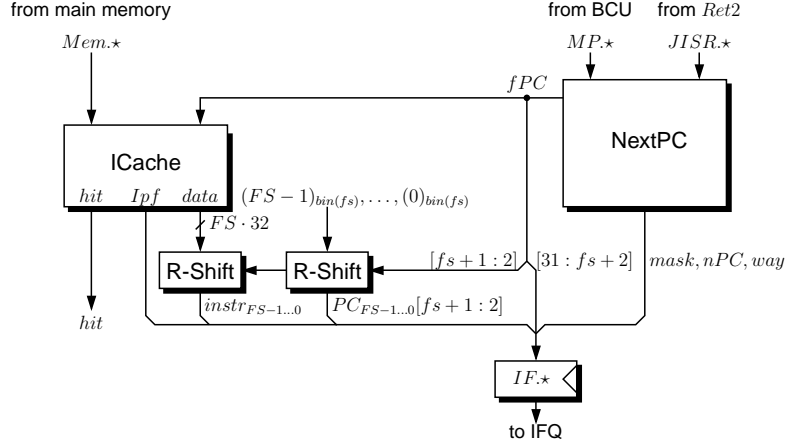
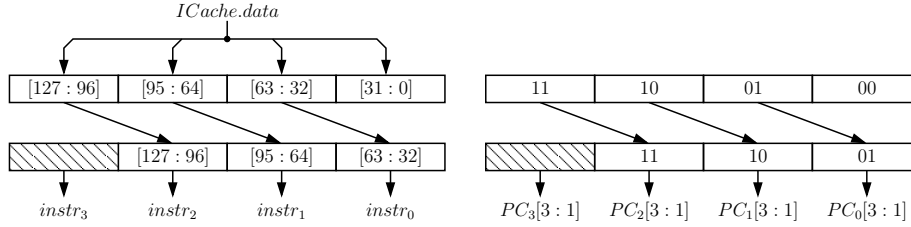


Figure 6.2: Instruction Fetch Unit

Figure 6.3: Instruction Shift for $fs = 2$ and $\langle fPC[fs+1:2] \rangle = 1$

that belongs to the current basic block. See figure 6.3 for an example of this shift for $fs = 2$ and $\langle fPC[fs+1:2] \rangle = 1$.

All instructions returned by the cache belong to the same (aligned) fetch block, thus the high order bits $[31 : fs+2]$ of the addresses of these instructions are identical. Let $PC_i[fs+1:2]$ denote the low order bits of the address of the instruction with index i . The value of $PC_i[fs+1:2]$ can be computed by adding the value of the bits $fs+1:2$ of the fetch-PC to i . Hence, the busses $PC_\star[fs+1:2]$ can be computed by shifting the constant vector $(FS-1)_{bin(fs)}, \dots, (0)_{bin(fs)}$ by $\langle fPC[fs+1:2] \rangle \cdot fs$ positions to the right.

To invalidate the instructions following the predicted branch position in the fetch block, the bus *mask* computed by the circuit *NextPC* is used. It encodes the number of instructions in the fetch block which belong to the current basic block in addition to the instruction addressed by the fetch-PC. The bus *mask* uses half-unary encoding, thus if in total j instructions of the fetch block belong to the current basic block, the bits $mask[j-1:0]$ are one, the bits $mask[FS-1:j]$ are zero. Note that the number of ones in the encoding is the number of valid instructions in the current fetch block as bit 0 of half-unary encodings is always one. Hence, the signal $mask[i]$ for $i \in \{0, \dots, FS-1\}$ can be used as valid signal for the instruction bus $instr_i$. Note that by shifting the output of the cache to the right, invalid data are shifted into the leftmost $\langle fPC[fs+1:0] \rangle$ instruction busses. Thus, the bus *mask* may at most encode the number $FS - \langle fPC[fs+1:0] \rangle - 1$.

The circuit *NextPC* gets updated by the misprediction bus MP_\star from the branch checking unit (see section 6.5) and the interrupt bus $JISR_\star$ computed by the retire

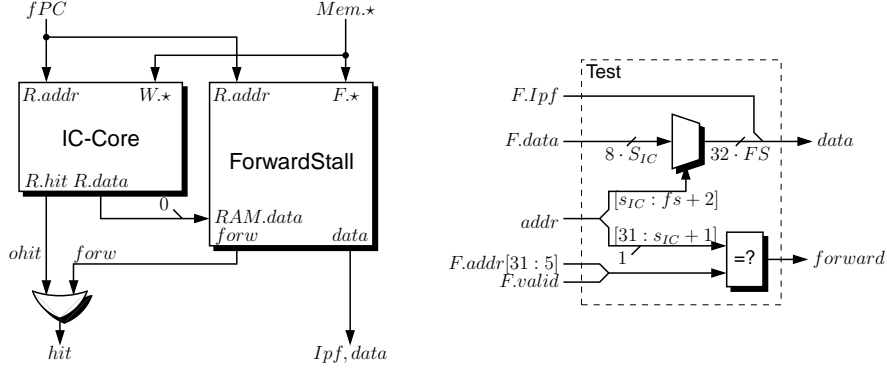


Figure 6.4: Instruction Cache

sub-phase *Ret2* (see section 4.5.4). The bus *MP.** contains information about all mispredicted branches, including instructions which have been wrongly predicted to be branches. Using the bus *JISR.**, the instruction fetch is updated in case of an interrupt.

The instruction fetch unit delivers apart from the actual instructions the addresses of the instructions, the predicted next fetch-PC, and the way of the branch target buffer (see section 6.2.3). These informations are needed by the branch checking unit (see section 6.5) and for interrupt handling (see section 4.5.3) and are sent along with the instructions through the processor.

6.2.2 Instruction Cache

The instruction cache is a simple blocking cache with $S_{IC} = 2^{s_{IC}}$ bytes per line ($S_{IC} \geq 4 \cdot FS$), $L_{IC} = 2^{l_{IC}}$ lines, and $K_{IC} = 2^{k_{IC}}$ ways, similar to the cache core of the data cache. Since instruction fetch is done in order a non-blocking cache has no advantages. The instruction cache is only updated by the main memory. Forwarding can be done by a forwarding circuit with stalling.

Figure 6.4 depicts the instruction cache. The cache core circuit *IC-Core* contains the cache RAM including the replacement circuit. The cache core is assumed to be not sectorized; the modifications for a sectorized cache core are simple, but not treated in this thesis. The page fault signal *Ipfc* is computed by the main memory. It can not be active if the data is already located in the cache core.

The sub-circuit *Test* of the forwarding circuit must be adopted if $S_{IC} > FS$. Then the requested fetch block is selected from the cache-line returned by the main memory. The same has to be done to the line returned by the cache core (not shown in the figure).

Note that the memory might return data from a fetch request that was started before the instruction fetch has been cleared due to a misprediction or an interrupt. Thus, the returned data may be not valid for the fetch. Thus, it is necessary that the address on the result bus of the main memory is checked for each stage of the forwarding circuit.

If neither the core nor the forwarding circuit return valid data ($ohit \vee forw = 0$) and the last stage of the forwarding pipeline contains a valid entry, the access is treated as a miss. In this case the stage (and the preceding stages) are stalled and a memory

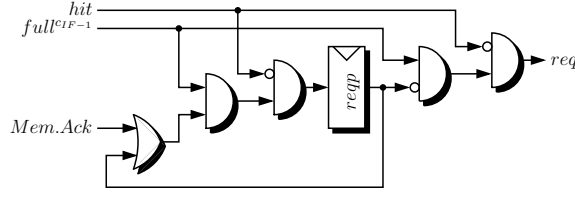


Figure 6.5: Instruction Cache Control

request is started. When the memory returns the data *forw* gets active and the valid data is sent to the output.

Figure 6.5 shows the computation of the request signal to the main memory. A new request to the main memory is started if a miss is in the last stage ($full^{c_{IF}-1} \wedge \overline{hit}$) and no request is pending ($reqp = 0$). As soon as the main memory does not stall, i.e. $Mem.Ack = 1$, the request is accepted and the register *reqp* is set. The register *reqp* stays set until the request returns and *hit* gets active. The address for the main memory request can be taken from the last stage of the forwarding circuit.

Cost and Delay

Let $L_{DC} = 2^{l_{DC}}$ be the number of lines, $S_{IC} = 2^{s_{IC}}$ be the number of bytes of a line, and $K_{IC} = 2^{k_{IC}}$ be the number of ways of the instruction cache. Let c_{IF} be the number of cycles of the instruction fetch. If an LRU algorithm is used for replacement, the cost and delay of the instruction cache can be computed similar to the cache core of the data cache. The delay and cost of such a cache are [MP00]:

$$\begin{aligned}
 D(\text{IC-Core}) &\leq \max\{D(\text{RAM}(L_{IC}, 8 \cdot S_{IC}, 1, 1)), \\
 &\quad D(\text{RAM}(L_{IC}, 33 - l_{IC} - s_{IC}, 1, 1)) \\
 &\quad + D(\text{EQ}(33 - l_{IC} - s_{IC}))\} \\
 &\quad + D(\text{Sel}(K_{IC})), \\
 C(\text{IC-Core-Replace}) &\leq C(\text{RAM}(L_{IC}, K_{IC} \cdot k_{IC}, 1, 1, c_{IF})) \\
 &\quad + K_{IC} \cdot (C(\text{EQ}(k_{IC})) + C_{AND}) + C(\text{PP-OR}(K_{IC})) \\
 &\quad + K_{IC} \cdot k_{IC} \cdot 5 \cdot C_{MUX} + C(\text{Enc}(K_{IC})) \\
 &\quad + \lceil (34 - l_{IC} + 2 \cdot k_{IC}) / 2 \rceil \cdot c_{IF} \cdot C_{REG}, \\
 C(\text{IC-Core}) &\leq K_{IC} \cdot (C(\text{RAM}(L_{IC}, 8 \cdot S_{IC}, 1, 1, c_{IF})) \\
 &\quad + C(\text{RAM}(L_{IC}, 33 - l_{IC} - s_{IC}, 1, 1, c_{IF})) \\
 &\quad + C(\text{EQ}(33 - l_{IC} - s_{IC}))) \\
 &\quad + 8 \cdot S_{IC} \cdot C(\text{Sel}(K_{IC})) + C(\text{OR-Tree}(K_{IC})) \\
 &\quad + C(\text{Dec}(k_{IC})) + K_{IC} \cdot C_{AND} \\
 &\quad + \begin{cases} 0 & \text{if } K_{IC} = 1 \\ C(\text{IC-Core-Replace}) & \text{if } K_{IC} > 1 \end{cases}.
 \end{aligned}$$

Let FS be the number of instructions per fetch block and let $c_{FS}(32 - s_{IC})$ be the number of cycles needed for forwarding with stalling for $32 - s_{IC}$ address bits. For

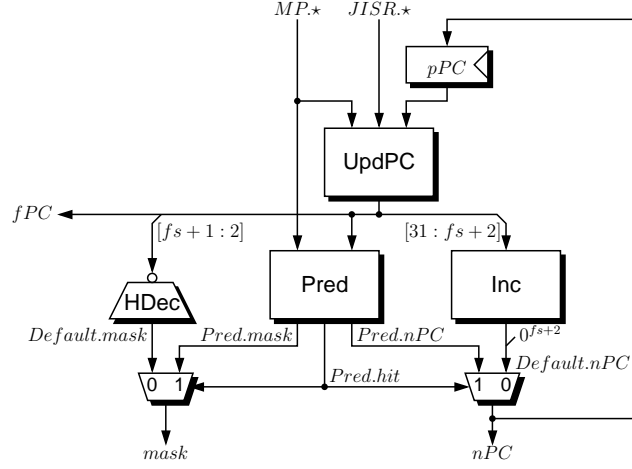


Figure 6.6: Next PC Circuit

the overall delay and cost of the instruction cache it holds:

$$\begin{aligned}
 D(\text{ICache}) &\leq D(\text{IC-Core}) + \max\{2 \cdot D_{AND}, D_{MUX}\} + D_{AND}, \\
 C(\text{ICache}) &\leq C(\text{IC-Core}) \\
 &\quad + C(\text{ForwardStall}(32 - s_{IC}, 8 \cdot S_{IC} + 1, c_{IF}, c_{FS}(32 - s_{IC}))) \\
 &\quad + D_{OR} + (c_{IF} + 1) \cdot 32 \cdot FS \cdot C(\text{Sel}(4 \cdot S_{IC}/FS)) \\
 &\quad + 5 \cdot C_{AND} + 2 \cdot C_{OR} + C_{REG}.
 \end{aligned}$$

6.2.3 Computation of the Next Fetch-PC

Overview

The circuit **NextPC** is divided into two parts. The first part computes the current fetch-PC fPC based on the predicted fetch-PC from the last cycle (pPC) and the input busses $MP.*$ and $JISR.*$. The second part computes the fetch-PC for the next cycle and the number of valid instructions in the current fetch block using branch prediction. The branch prediction is based on the value of fPC computed in the first part.

Figure 6.6 shows the circuit **NextPC**. The fetch-PC fPC is computed in the sub-circuit **UpdPC**. If an interrupt has occurred (indicated by $JISR.full = 1$), fPC is set to the start of the interrupt service routine $JISR.sisr$. Otherwise, if a misprediction has been detected by the BCU (indicated by $MP.full$), fPC is set to the corrected branch target $MP.cPC$. If neither an interrupt occurred nor a misprediction has been detected the address predicted in the last cycle which is saved in the register PC is used. Thus, the circuit **UpdPC** computes:

$$fPC = \begin{cases} JISR.sisr & \text{if } JISR.full \\ MP.cPC & \text{if } \overline{JISR.full} \wedge MP.full \\ pPC & \text{if } \overline{JISR.full} \wedge \overline{MP.full} \end{cases}$$

The fetch-PC is used in the circuit **Pred** to predict the next fetch-PC nPC and the bus $mask$ which encodes the number of valid instruction in the current fetch block. If

the prediction circuit does not produce a valid prediction (indicated by $Pred.hit = 0$) default values are used for nPC and $mask$. The default values are based on the assumption that the fetch block does not contain any branch instructions following the address fPC . Accordingly, nPC is set to the beginning of the next cache-line and all instructions following the fetch address are assumed to be valid. The default value for the next fetch-PC is:

$$default.nPC[31 : 0] = (\langle fPC[31 : fs + 2], 0^{fs+2} \rangle + 2^{fs+2})_{bin(32)}.$$

The bus $mask$ must encode the number of valid instructions in the current fetch block in half-unary encoding (not counting the instruction addressed by the fetch PC). The current fetch block contains $FS - \langle fPC[fs + 1 : 2] \rangle - 1$ instructions that succeed the instruction addressed by the fetch-PC. Thus, the default value for the bus mask can be computed as:

$$\begin{aligned} default.mask[fs - 1 : 0] &= (FS - \langle fPC[fs + 1 : 2] \rangle - 1)_{hun(FS)} \\ &= (\overline{\langle fPC[fs + 1 : 2] \rangle} \pmod{FS})_{hun(FS)} \\ &= (\overline{\langle fPC[fs + 1 : 2] \rangle})_{hun(FS)}. \end{aligned}$$

Hence, the default values can be computed using a half-decoder and an incrementer.

Let $FS = 2^{fs}$ be the size of the fetch block in words. The cost and delay of the circuit $nextPC$ are:

$$\begin{aligned} D(fPC) &\leq D(UpdPC) \leq 2 \cdot D_{MUX}, \\ D(NextPC) &\leq D(fPC) + \max\{D(HDec(fs)), D(Pred), D(Inc(30 - fs))\} \\ &\quad + D_{MUX}, \\ C(UpdPC) &\leq 2 \cdot 32 \cdot C_{MUX}, \\ C(NextPC) &\leq C(HDec(fs)) + C(Inc(30 - fs)) + C(Pred) + C(UpdPC) \\ &\quad + (32 + FS) \cdot C_{MUX} + 32 \cdot C_{REG}. \end{aligned}$$

Prediction Circuit

The circuit $Pred$ does the actual branch prediction. In this thesis a simple branch target buffer (BTB) [LS84] is described. It is a cache which saves the last target address of jumps and taken branches. More sophisticated algorithms using a return stack [Web88] or combining multiple prediction schemes [McF93] lie beyond the scope of this thesis.

For not-taken branches the fetching must continue at the next instruction. Thus, regarding the address of the next instruction not-taken branches can be treated like non-branch instructions. To exploit this fact the BTB does not contain entries for not-taken branches. The BTB predicts the target of the first branch instruction in the current fetch block which is a predicted to be taken branch. Hence, the prediction circuit can correctly predict up to FS branch instructions in one cycle if the first $FS - 1$ instructions are not-taken branches.

The branch prediction reads the BTB every cycle in order to check whether it contains information about the current fetch-PC. On a hit the BTB returns the predicted values for $mask$ and nPC and sets the signal hit to one. Otherwise the signal hit is set to zero.

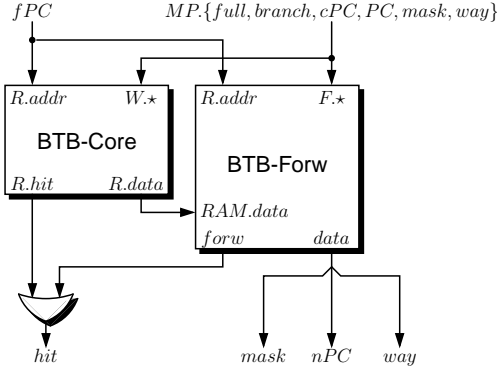


Figure 6.7: Prediction circuit

The update of the BTB is controlled by the branch checking unit. The simple BTB used here needs to be updated only in case of a misprediction. More complex prediction schemes also learn from correctly predicted branches, thus those schemes update the BTB for every branch instruction.

Two different cases of mispredictions must be kept apart. If the result of a branch was predicted wrongly, the corrected result is saved in the branch target buffer. If the branch target buffer contains an entry for a non-branch instruction or a branch that should be predicted to be not taken, the entry must be invalidated. Note that the first case may occur due to self-modifying code.

The BTB is built as a $K_{BTB} = 2^{k_{BTB}}$ way cache with $L_{BTB} = 2^{l_{BTB}}$ lines of width $32 + FS$ (nPC and $mask$) and can be built similar to the instruction cache. Figure 6.7 shows the prediction circuit Pred containing the BTB. The memory bus used in the instruction cache is replaced by the bus $MP.*$. In contrast to the instruction cache a miss does not produce a stall or a memory request.

The BTB has to be updated if the branch checking unit returns a misprediction ($MP.full$ is active). The signal $MP.branch$ indicates if the predicted branch was indeed a branch. This signal is used to write the valid bit of the cache entry. The fields $mask$ and nPC of the entry are updated with the values of the busses $MP.mask$ and the correct branch target $MP.cPC$. The bus $MP.fPC$ contains the fetch-PC of the mispredicted branch and is used to select the line and update the cache directory.

The prediction circuit also computes returns way. If hit is active the way addresses the BTB entry that caused the hit and that has to be updated in case of a misprediction. If hit is inactive the way indicates which entry has to be updated if a new entry is made. Since the updates of the BTB are started by the BCU, the way must be sent to the BCU along with the instruction. The BCU then returns the way on the bus $MP.way$ that determines, which way is written on an update of the BTB.

Pipelining of the circuit is not possible as the predicted PC nPC has to be computed in one cycle of the slow clock $IFCLK$. However, for performance reasons forwarding is used in order to instantly use the results of the BCU. To support updates and requests at the same time all RAM blocks have separated read and write ports.

If a basic block is distributed over multiple fetch blocks, the fetch-PC of the branch is always the address of the fetch block containing the branch. Hence, the probability that the last $fs + 2$ bits of the fetch-PC are all equal to zero is disproportionately high.

For this reason the bits $[fs + 3 : 4]$ are used to address the cache-lines instead of the usual bits $[fs + 1 : 2]$. This usually results in a better prediction, as shown in [Yeh93].

If $K_{BTB} = 2^{k_{BTB}}$ and $L_{BTB} = 2^{l_{BTB}}$ are the number of ways and lines of the BTB and $FS = 2^{fs}$ is the fetch size, the cost and delay of the core of the BTB can be estimated as:

$$\begin{aligned}
D(\text{BTB-Core}) &\leq \max\{D(\text{RAM}(L_{BTB}, 32 + fs, 1, 1)), \\
&\quad D(\text{RAM}(L_{BTB}, 33 - l_{BTB}, 1, 1)) \\
&\quad + D(\text{EQ}(33 - l_{BTB}))\} \\
&\quad + D(\text{Sel}(K_{BTB})), \\
C(\text{BTB-Core-Replace}) &\leq C(\text{RAM}(L_{BTB}, K_{BTB} \cdot k_{BTB}, 1, 1, 1)) \\
&\quad + K_{BTB} \cdot (C(\text{EQ}(k_{BTB})) + C_{AND}) \\
&\quad + C(\text{PP-OR}(K_{BTB})) \\
&\quad + K_{BTB} \cdot k_{BTB} \cdot 5 \cdot C_{MUX} + C(\text{Enc}(K_{BTB})) \\
&\quad + \lceil (34 - l_{BTB} + 2 \cdot k_{BTB}) / 2 \rceil \cdot C_{REG}, \\
C(\text{BTB-Core}) &\leq K_{BTB} \cdot (C(\text{RAM}(L_{BTB}, 32 + FS, 1, 1, 1)) \\
&\quad + C(\text{RAM}(L_{BTB}, 33 - l_{BTB}, 1, 1, 1)) \\
&\quad + C(\text{EQ}(33 - l_{BTB}))) \\
&\quad + (32 + FS) \cdot C(\text{Sel}(K_{BTB})) + C(\text{OR-Tree}(K_{BTB})) \\
&\quad + C(\text{Dec}(k_{BTB})) + K_{BTB} \cdot C_{AND} \\
&\quad + \begin{cases} 0 & \text{if } K_{BTB} = 1 \\ C(\text{BTB-Core-Replace}) & \text{if } K_{BTB} > 1 \end{cases}.
\end{aligned}$$

The overall delay and cost of the prediction circuit are:

$$\begin{aligned}
D(\text{Pred}) &\leq D(\text{BTB-Core}) + \max\{2 \cdot D_{AND}, D_{MUX}\}, \\
C(\text{Pred}) &\leq C(\text{BTB-Core}) + C_{OR} \\
&\quad + C(\text{ForwardStall}(32, 33 + FS + k_{BTB}, 1, c_{FS}(32))).
\end{aligned}$$

6.2.4 Instruction Fetch Control

The instruction fetch control (see figure 6.8) computes the stall signals for the instruction fetch unit. Note that the branch prediction done in the circuit **NextPC** must be done within one cycle. However, the access to the instruction cache can be pipelined as the prediction does not depend on the result of the cache access. Thus, the instruction fetch unit can have multiple stages.

Additionally the instruction fetch control synchronizes between the fast clock CLK and the slower clock $IFCLK$. Let c_{IF} be the number of stages of the instruction fetch. The synchronization of the clocks is done via the full signal $IF.full^{c_{IF}}$ for the output registers $IF.\star$ of the instruction fetch unit (see figure 6.2). The full signal $IF.full^{c_{IF}}$ is updated every cycle of the fast clock CLK . It is set to one if at the end of the last CLK cycle before the rising edge of $IFCLK$ (indicated by $lastcycle = 1$) the instruction cache returns a hit and the preceding stage is full, i.e. $IF.full^{c_{IF}} \wedge IC.hit = 1$.

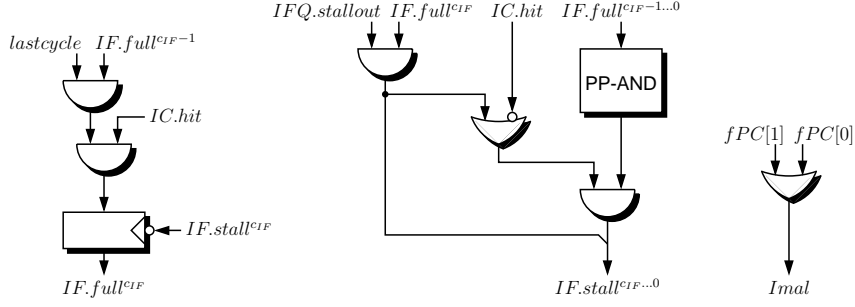


Figure 6.8: Instruction Fetch Control

The output registers of the instruction fetch unit are stalled as long as the full signal $IF.full^{c_{IF}}$ is active and the IFQ cannot accept new data ($IFQ.stallOut = 1$). Note that full signal $IF.full^{c_{IF}}$ is clocked with the fast clock CLK . Thus, as soon as the stall output of the IFQ $IFQ.stallOut$ becomes zero, the signal $IF.full^{c_{IF}}$ is reset in order to mark that the instructions in the output registers of the instruction fetch unit have been sent to the IFQ. Then the output registers can be updated with the next rising edge of the slow clock $IFCLK$. The last stage of IF can generate a stall if the instruction cache returns a miss. The stall computation is adapted as usual.

The instruction fetch control also computes the interrupt signal $Imal$ indicating a misaligned fetch. It can be computed by OR-ing the two least significant bits of the fetch address.

6.2.5 Cost and Delay

The new value for the register PC has to be computed in one cycle. Thus, the delay of the circuit $NextPC$ gives a lower bound for the cycle time $m \cdot \tau$ of the instruction fetch clock $IFCLK$. Thus, it must hold:

$$m \cdot \tau - 5 \geq D(NextPC).$$

The overall delay of the instruction fetch and the number of cycles c_{IF} needed for the instruction fetch are:

$$D(IF) \leq \max(D(fPC) + D(IC) + fs \cdot D_{MUX}, D(NextPC)),$$

$$c_{IF} = \lceil D(IF) / (m \cdot \tau - 5) \rceil.$$

The width of the registers $IF.\star$ is $67 + FS \cdot (33) + k_{BTB}$ (full bit, two interrupts, fetch-PC, predicted PC, FS instructions including valid bit, and the way of the BTB). Thus, the cost of the instruction fetch is:

$$C(IF) \leq C(IC) + C(NextPC) + (8 \cdot S_{IC} - FS) \cdot C_{MUX}$$

$$+ (67 + FS \cdot 33 + k_{BTB}) \cdot C_{REG}.$$

6.3 Instruction Fetch Queue

The instruction fetch queue (IFQ) serializes the parallel instruction stream delivered by the instruction fetch unit. It is a parallelly loadable FIFO queue with FS entries.

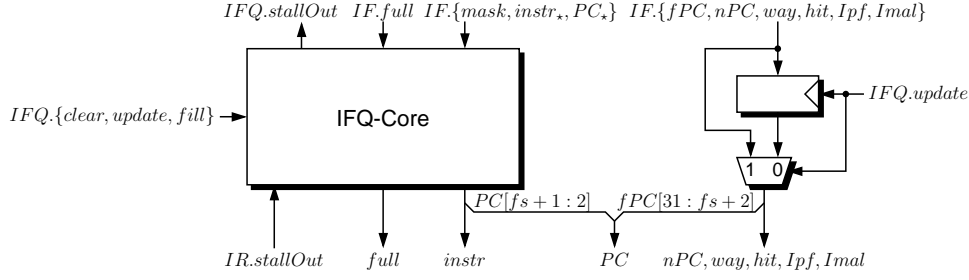
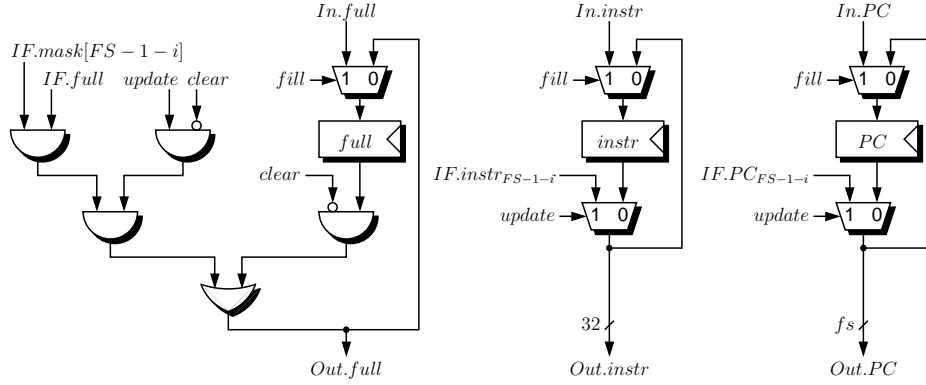


Figure 6.9: Instruction fetch queue

Figure 6.10: IFQ entry i

New instructions are only filled into the IFQ if the queue is empty. This simplifies the design of the IFQ as an instruction $instr_i$ for $i \in \{0, \dots, FS - 1\}$ is always filled into the entry number $FS - 1 - i$. Also all valid instructions in the queue share the signals $fPC[31 : fs + 2]$, nPC , way , hit , Ipf , and $Imal$ are the same, since these signals are the same within a fetch block.

Figure 6.9 depicts an overview of the instruction fetch queue. The signals $fPC[31 : fs + 2]$, nPC , way , hit , Ipf , and $Imal$ are stored only once for all entries of the queue. They are updated whenever new instructions are loaded into the queue (i.e., the queue is empty). The instructions $instr_*$ and the lower order bits of the instruction's address $PC_*[fs + 1 : 2]$ are stored in the sub-circuit IFQ-Core, which is built similar to the other queues, e.g., the reservation station. This circuit also computes the stall output of the IFQ $IFQ.stallOut$, and the full bit which is sent to the instruction register environment. The address PC of the instruction sent to the decode phase can be obtained by concatenating the common high order bits $fPC[31 : fs + 2]$ and the low order bits $PC[fs + 1 : 2]$ of the instruction.

6.3.1 IFQ Entries

A single entry of the circuit IFQ-Core is shown in figure 6.10. New instructions and addresses are filled into the IFQ when the signal $update$ is active. The instruction $FS - 1 - i$ is then filled into entry i for $i \in \{0, \dots, FS - 1\}$. The instruction is valid if the signal $IF.mask[FS - 1 - i]$ is active and the output of the instruction fetch unit is valid (indicated by $IF.full$).

If the signal $fill$ is active, the content of the entry advances into the next entry.

Since the IFQ is only loaded parallelly the input $In.full$ of entry 0 has to be constantly zero. The signal *clear* clears all entries of the IFQ, even if new entries are filled into the IFQ.

6.3.2 Control

The instruction fetch aligns all valid instructions within a fetch block by shifting them to the right (see section 6.2.1). The rightmost instruction $instr_0$ is filled into entry $FS - 1$. Therefore, no invalid instruction is followed by a valid one inside the queue. Thus, it is not necessary to remove empty entries from the queue. Hence, if the instruction register produces a stall ($IR.stallout = 1$), all entries of the queue are stalled. Thus, for each entry $i \in \{0, \dots, FS - 1\}$ of the IFQ it holds:

$$IFQ.fill = \overline{IR.stallout}.$$

Note that in contrary to the reservation stations the fill signal directly depends on the stall input. This means that a new fetch block may be loaded into the queue, and simultaneously the fill bit is active, i.e. the queue instructions in the queue advance. Thus, if the instruction register environment does not produce a stall while the IFQ is loaded, the instruction $instr_0$ is directly sent to the instruction register. The instruction $FS - 1 - i$ is then filled into entry $i + 1$.

Since no full entry may follow an empty one in the IFQ, the whole queue is empty if the last entry (number $FS - 1$) is empty. New instructions are filled into the IFQ if the queue is empty. The instruction fetch must be stalled if the IFQ is not empty. Hence,

$$\begin{aligned} IFQ.empty &= full_{FS-1}, \\ IFQ.update &= full_{FS-1}, \\ IFQ.stallOut &= \overline{full_{FS-1}}. \end{aligned}$$

The decode phase also needs the signal $IFQ.pb$ which indicates whether an instruction was predicted to be a taken branch (see appendix C.2.1). This is the case if the hit signal of the prediction circuit was active and the instruction is the last valid instruction in the fetch block. Thus:

$$IFQ.pb = IFQ.hit \wedge Entry_{FS-1}.full \wedge \overline{Entry_{FS-2}.full}.$$

6.3.3 Cost and Delay

The critical path in the IFQ is the update of the full bits of the entries. This path bounds δ to be at least

$$\delta \geq 2 \cdot D_{AND} + D_{OR} + D_{MUX}. \quad (6.1)$$

The cost of the IFQ entries and the IFQ control are:

$$\begin{aligned} C(\text{IFQ-Entry}) &\leq (65 + 2 \cdot FS) \cdot C_{MUX} + (33 + FS) \cdot C_{REG} \\ &\quad + 4 \cdot C_{AND} + C_{OR}, \\ C(\text{IFQ-Control}) &\leq 2 \cdot C_{AND} + C_{OR}. \end{aligned}$$

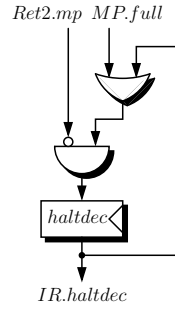


Figure 6.11: Instruction Register Control

The total cost of the IFQ is:

$$C(\text{IFQ}) \leq FS \cdot C(\text{IFQ-Entry}) + C(\text{IFQ-Control}) \\ + (67 - FS + k_{BTB}) \cdot (C_{REG} + C_{MUX}).$$

This cost is added to the cost of the of instruction fetch unit.

6.4 Instruction Register Environment

The decoding of instructions must be stopped if a branch misprediction has been detected. Instruction decoding may be resumed with the correct instructions when the mispredicted instruction has retired. This is done by the instruction register control (see figure 6.11).

The register *haltdec* is set if a misprediction occurred (i.e., *MP.full* is active). If *haltdec* is active, the instruction register is stalled (see section 4.1.7). The register *haltdec* is reset again if the mispredicted instruction leaves the second retire phase. This is indicated by the signal *Ret2.mp*.

The cost of the instruction register control is added to the cost of the decode sub-phase *D1*:

$$C(D1)^+ \leq C_{REG} + C_{AND} + C_{OR}.$$

6.5 Branch Checking Unit

The branch checking unit (BCU) computes the target of branch instructions and checks if the branch prediction has correctly predicted that target. In order to check the prediction every instruction that has been predicted to be a branch instruction is sent to the BCU, even if the instruction is actually no branch instruction (see section 4.1.3). In case of a misprediction, the BCU initiates a rollback and updates the branch target buffer. For jump-and-link and return-from-exception instruction the BCU also computes a result that has to be written into the register files.

The BCU is divided into the two circuits **BComp** and **BCheck**. The circuit **BComp** computes the result and the branch target of a branch instruction. The circuit **BCheck** uses the outputs of **BComp** to check whether the target of the branch instruction has been predicted correctly.

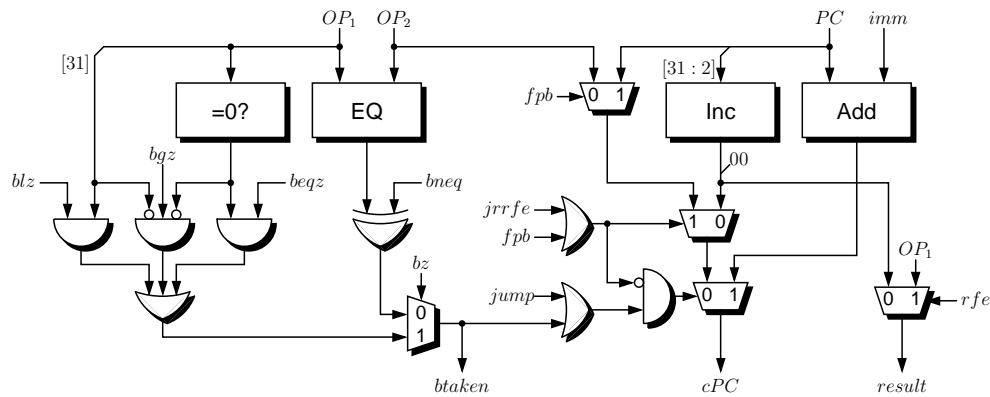


Figure 6.12: Branch Computation

The circuit BComp is shown in figure 6.12. The circuit uses multiple control signals which can be computed from the opcode of the branch instruction. The computation of these signals is not assumed to be critical and not discussed in detail. The left part of the circuit computes the signal *btaken* which indicates if a branch has to be taken. Using the signal *btaken* the corrected PC *cPC* of the branch instruction is computed in the right part of the circuit.

Two types of branches are supported: regular branches which check if the two operands OP_1 and OP_2 are equal and branch-zero instructions (indicated by the signal *bz*) which compares the first operand against zero. Regular branches can be divided into branch-equal and branch-not-equal instructions (indicated by *bneq*). The type of a branch-zero instruction is defined by the signals *blz*, *bqz*, and *beqz* indicating a branch-less-than-zero, branch-greater-than-zero, respectively branch-equal-zero. Multiple of these signals may be active to indicate, e.g., a branch-greater-equal-zero.

Note that branch on floating point condition code (BC1) instructions (see table A.7 in appendix A) use the special register *FCC* as first operand. Thus, the instruction BC1F can be implemented by setting *beqz* to one, the instruction BC1T by setting *bqz* to one.

The correct PC of the instruction following the branch instruction *cPC* can have 4 different values:

- The signal *f pb* (computed in the decode phase) is active if the instruction is not a branch instruction but has been wrongly predicted to be a branch instruction by the branch prediction. The instruction fetch must then be restarted at the wrongly predicted branch, i.e., *cPC* must be set to the address of the instruction *PC*.
- For jump-register and return-from-exception instructions (indicated by *jrr fe*) the bus *cPC* must be set to the value of the second operand.
- For jump instructions (indicated by *jump*) and taken branches (indicated by *btaken*), an immediate constant is added to the address of the instruction to compute *cPC*.
- For not-taken branches the bus *cPC* is the by 4 incremented address of the branch instruction.

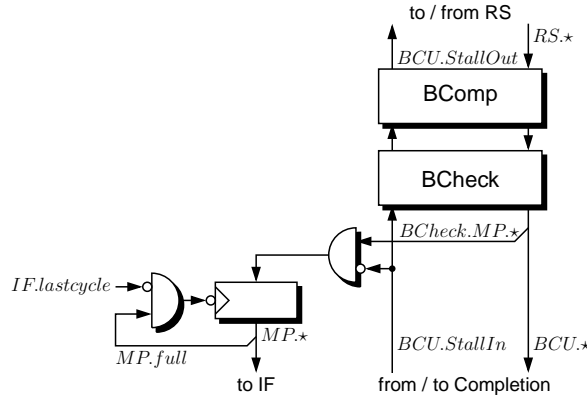


Figure 6.14: Branch Checking Unit

6.5.1 Stall Computation

If the BCU detects a misprediction, it must update the instruction fetch unit. The IFU can only accept new data if the signal *lastcycle* is active. In order not to affect the completion of correctly predicted branches an extra stage is added in which mispredicted branches wait until the signal *lastcycle* gets active (see figure 6.14). If this stage contains a mispredicted branch, the BCU is cleared.

A branch instruction proceeds to the additional stage at the same time it enters the completion stage. If the branch instruction cannot proceed to the completion phase the additional stage must not be filled. Otherwise if the branch instruction was mispredicted, the BCU would be cleared and the branch instruction does not complete, which is needed for the rollback.

This additional stage does not have to stall the other stages of the BCU as all stages of the BCU are cleared anyway if a misprediction is in the additional stage (indicated by the signal *BCU.mp*). Thus, the other stages of the BCU are only stalled by the signal *BCU.stallIn* received from the completion phase. The additional stage is stalled by

$$MP.full \wedge \overline{IF.lastcycle}.$$

Note that once a misprediction is in the additional stage, the stage is stalled until the instruction fetch unit can be updated. Thus, the mispredicted branch in the additional stage will not be overwritten by succeeding instructions.

6.5.2 Cost and Delay

The delay of the BCU is:

$$\begin{aligned} D(btaken) &\leq \max\{D(EQ(32)) + D_{XOR}, D(Zero(32)) + 2 \cdot D_{AND} + D_{OR}\} \\ &\quad + D_{MUX}, \\ D(BCU) &\leq \max\{D(Add(32)), D(Inc(32)) + D_{MUX}, \\ &\quad D(btaken) + D_{AND} + D_{OR}\} \\ &\quad + D_{MUX} + D(EQ(32)) + D_{OR} + D_{AND}. \end{aligned}$$

Let the variable b_{BCU} be one if a buffer circuit is added after the input registers of the BCU, otherwise b_{BCU} is zero. The number of cycles c_{BCU} needed for the BCU is:

$$c_{BCU}(b_{BCU}) = \lceil (D(\text{BCU}) + b_{BCU} \cdot D_{MUX}) / \delta \rceil.$$

Let $BCU.stallIn$ be the stall input of the BCU from the completion phase. If $b_{BCU} = 0$, the delay of the output stall signal $stallOut$ of the BCU is:

$$D(stallOut) \leq \max\{D(\text{AND-Tree}(c_{BCU}(0) + 1)), D(BCU.stallIn)\} + D_{AND}.$$

Let e_{RS_7} be the number of entries of the reservation station of the BCU. The reservation station then requires (see equation (4.15) on page 62):

$$\delta \geq D(BCU.stallOut) + \begin{cases} 2 \cdot D_{AND} & \text{if } e_{RS_7} = 1 \\ 3 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS_7} > 1 \end{cases}.$$

If this equation can not be fulfilled, b_{BCU} must be one. This reduces the bound to:

$$\delta \geq \max\{D(\text{AND-Tree}(c_{BCU}(1) + 2)), D(BCU.stallIn)\} + D_{AND}$$

which hold true for $\delta \geq 5$. The requirement for the input stall signal of the reservation station are then fulfilled by construction of the reservation station.

The number of inputs of the BCU is $161 + k_{BTB} + l_{ROB}$, the number of outputs is $97 + fs + k_{BTB} + l_{ROB}$. The cost of the BCU is approximated by:

$$\begin{aligned} C(\text{BCU}) \leq & C(\text{Add}(32)) + C(\text{Inc}(32)) + 2 \cdot C(\text{EQ}(32)) + C(\text{Zero}(32)) \\ & + 129 \cdot C_{MUX} + 9 \cdot C_{AND} + 7 \cdot C_{OR} + C_{XOR} \\ & + (32 + fs + k_{BTB}) \cdot C_{REG} + 2 \cdot C_{AND} \\ & + (c_{BCU} - 1) \cdot \lceil (258 + fs + 2 \cdot k_{BTB} + 2 \cdot l_{ROB}) / 2 \rceil \cdot C_{REG}. \end{aligned}$$

6.6 Processor Flush

The processor must be flushed if an interrupt occurred or a misprediction has been detected. Interrupts are detected during the retire sub-phase *Ret2*, mispredictions are detected by the BCU. After a flush the processor must be in a state it would have after the execution of the instruction that caused the flushing (respectively before the execution depending on the type of the interrupt or misprediction) if all instructions would be executed sequentially. This is done when the instruction enters the retire sub-phase *Ret3* since then all registers have the correct value and the producer tables can be reset.

In order to reduce the penalty of mispredictions, the instruction fetch is restarted at the correct address as soon as the misprediction is detected by the BCU. To guarantee that no wrongly fetched instruction causes another misprediction the flush of the processor is done in two steps. As soon as a misprediction is detected all succeeding branches are flushed. This is done by clearing the instruction fetch unit, the instruction fetch queue, the decode phase, and the BCU including the corresponding reservation station. Since these parts of the processor execute instructions in-order no preceding instruction is cleared. After the clear the instruction fetch is resumed at the correct

address, but the instruction succeeding the mispredicted instruction is stalled at the instruction register (see section 6.4). When the mispredicted instruction enters the retire sub-phase *Ret3* the remaining parts of the processor are flushed and the succeeding instruction can be decoded.

Interrupts are detected at the end of the retire sub-phase *Ret2*. Therefore, the flush of the processor is not done in two steps. When an instruction that caused an interrupt enters the retire sub-phase *Ret3* all parts of the processor are cleared.

Note that in parallel to first part of the flush due to a misprediction of the the flush due to an interrupt the instruction fetch unit must be updated. This can only be done if the signal *lastcycle* is active. The flushing must not affect the additional stage of the BCU and the first stage of the retire sub-phase *Ret3* since these stages contain the instructions that caused the flush. Note that interrupts have a higher priority than mispredictions. Thus, if a mispredicted instruction is in the additional stage of the BCU at the same time an instruction that caused an interrupt is in the first stage of the sub-phase *Ret3* the misprediction has no effect.

Also the remaining stages of the sub-phase *Ret3* must not be flushed as they contain instructions preceding the instruction that caused the flush. However, the writes to the producer table during *Ret3* are overwritten by the clear signal of the producer table. It follows:

$$\begin{aligned}
 & \left. \begin{array}{l} IF.clear \\ IFQ.clear \\ D1.clear \\ D2.clear \\ RS_6.clear \\ FU_6.clear \end{array} \right\} = (JISR.full \vee MP.full) \wedge \overline{lastcycle}, \\
 & \left. \begin{array}{l} RS_{0...5}.clear \\ FU_{0...5}.clear \\ Complete.clear \\ Ret1.clear \\ Ret2.clear \\ ROB.clear \\ PT.clear \end{array} \right\} = (JISR.full \wedge \overline{lastcycle}) \vee Ret2.mp.
 \end{aligned}$$

The clear signals are assumed to have a delay of zero. This is not a problem for the signals *JISR.full*, *Ret2.mp*, and *lastcycle* as they are not assumed to be critical. The signal *MP.full* is the critical signal of the BCU, but the OR- and AND-gate for the computation of the clear signal can be put in the additional stage for mispredicted branches and therefore have no impact on the delay of the BCU.

Chapter 7

Discussion

This chapter discusses the results of the preceding chapters. In section 7.1 the reduction of the stage depth below five gate delay is investigated. Section 7.2 discusses the advantages and disadvantages of the gate model used in this thesis. The cost and delay values of the $DLX_{\pi+}$ that can be derived from the formulas in the preceding chapters are presented in section 7.3 and compared against the Tomasulo DLX from [Krö99] on which the $DLX_{\pi+}$ is based. Related work is discussed in section 7.4

7.1 Stage depths below 5

In this thesis all circuits of the $DLX_{\pi+}$ were designed in order to allow a stage depth $\delta \geq 5$ for the given gate model. This section describes possible enhancements in order to allow a stage depth of 4.

All basic bounds for δ regarding forwarding and stalling hold for $\delta = 4$. If a stage can generate a stall the stall input may have at most delay see equation (2.10) on page 18)

$$D(stallIn) \leq \delta - (D_{OR} + D_{AND}).$$

If a stall signal is used in a forwarding circuit with stalling it must hold:

$$D(stall) \leq \delta - D_{MUX}.$$

Since the delay of stall signals can be reduced to D_{AND} by inserting buffer circuits these equations can be satisfied with $\delta = 4$. The forwarding circuits require (see equation (2.12) on page 23)

$$\delta \geq \max\{D_{MUX}, D_{OR}\} + D_{MUX},$$

which is fulfilled for $\delta = 4$.

The equation (4.14) on page 59 bounds δ for a reservation station of type i as follows:

$$\delta \geq \begin{cases} 3 \cdot D_{AND} & \text{if } e_{RS} = 1 \\ \max\{D_{AND}, D(\text{FFO}(e_{RS}))\} + 2 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS} > 1 \wedge i \notin \{0, 6\} \\ \max\{D_{AND}, D(\text{FFO}(e_{RS}))\} + 3 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS} > 1 \wedge i \in \{0, 6\} \end{cases}.$$

This equation can be fulfilled for $\delta = 4$ only if $e_{RS} = 1$, i.e. all reservation stations may have at most one entry. The number of functional units n_i of a type i for a given δ is bounded by equation (4.2) on page 43:

$$\delta \geq D_{MUX} + \max\{D_{OR}, \max_{0 \leq i \leq 6} (D(\text{PP-OR}(e_{RS_i})) + D(\text{FFO}(n_i)))\} \\ + \begin{cases} 0 & \text{if } n_i = 1 \\ D_{AND} & \text{if } n_i > 1 \end{cases}.$$

Hence, for $\delta = 4$ the number of functional units of for each type may be at most 2.

The issue circuit also requires that in one cycle an instruction can be issued to at least two groups (see equations (4.3) and (4.4) on page 44):

$$\delta \geq D_{AND} + D(\text{Sel}(2)) + D_{AND}, \\ \delta \geq \max\{2 \cdot D_{AND}, D_{AND} + D_{MUX}, D_{MUX}\} + D(\text{Sel}(2)).$$

Both equations hold true for $\delta \geq 4$. The requirements for the remaining stall signals of the decode phase can be reduced by adding additional buffer circuits. Note that this would mean to add a buffer circuit into a RAM block (see figure 2.7) which is not necessary for $\delta \geq 5$.

The completion phase can be built for $\delta = 4$ if a tree of two-port arbiters is used. These arbiters reduce the requirements of the completion phase to the bound given by equation (4.20) on page 68:

$$\delta \geq 2 \cdot D_{AND} + 2 \cdot D_{OR}$$

which also holds true for $\delta = 4$.

The retire phase has no additional requirements to the stage depth δ . However, the ROB environment used in the retire phase introduces multiple bounds on delta. The delay of the output *forwOut* of the forwarding circuits for the retire context of the ROB access is bounded by (see equation (4.24) on page 83)

$$D(\text{forwOut}) \leq \delta - (D_{MUX} + D_{OR} + D_{AND}).$$

In the standard implementation of the forwarding circuit the output *forwOut* has at least delay D_{OR} and thus the equation does not hold for $\delta = 4$. In order to fulfill the bound, the signal *forwOut* must be derived from the register $\text{forw}^{c_{Ret1}-1}$ of the last stage of the forwarding circuit (see figure 4.27 on page 82) and not from the signal $\text{forwUpd}^{c_{Ret1}-1}$ which has a delay of at least D_{OR} . This does not affect the correctness but increases the number of cycles c_{C2R} and c_{A2R} needed to forward from the write access in the completion- respectively allocation-context to the read access in the retiring-context by one.

The delay optimizations for the ROB control signal *headce* and *tailce* in section 4.6.7 also apply for $\delta = 4$. They only assume that equation (4.24) holds true and the delay of the signal $D1.\text{stallIn}^0 \vee D1.\text{genStall}^0$ is at most δ , which can be achieved by inserting buffer circuits in the decode phase (see above).

The counters for the full and empty bits require (see equation (4.27) on page 86)

$$\delta \geq 2 \cdot D_{MUX} + D_{AND}$$

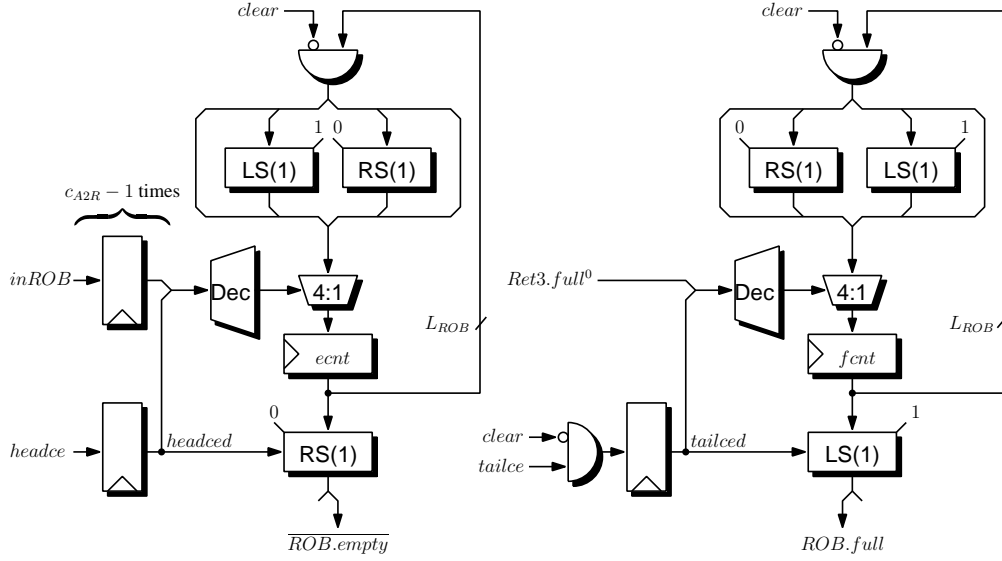


Figure 7.1: Optimized full and empty counter for $\delta = 4$

due to the loop for the computing of the next counter value. This does not hold for $\delta = 4$. Therefore, the counters for the full and empty bit must be modified as depicted in figure 7.1. Both counter delay one of the control signals for incrementing and decrementing (*tailce* respectively *headce*) by one cycle and use the delayed signal to modify the output of the counter (similar to the counter for the full bit in figure 4.31 on page 91). Due to the delaying of the signals *tailce* and *headce* all control signals for the counter come directly out of a register. This allows to use a 4 input selection circuit instead of two variable shifters to compute the next value of the counter. Based on the value of the increment respectively decrement signals the selection circuit selects between the shifted or the un-shifted counter value. If only one of the increment or decrement signals is active, the left- respectively right-shifted value is selected. If none or both signals are active the un-shifted counter value is selected, since the left- and the right-shift eliminate each other. Note that the constant shifters in figure 7.1 only consists of wiring and therefore have no delay. Thus, the modified counter only bounds delta by:

$$\delta \geq \max\{D(\text{Dec}(2)), D_{AND}\} + D(\text{Sel}(4))$$

which holds true for $\delta = 4$.

The update queue of the data cache bounds δ as follows (see equation (5.12) on page 135):

$$\delta \geq \max\{D(\text{OR-Tree}(e_{UQ})), D_{AND}\} + 3 \cdot D_{AND}.$$

Thus, for $\delta = 4$ the number of entries of the update queue e_{UQ} must be one. If the optimized completion for stores is used, the path from the stall input to the full bit of the entry bounds the stage depth by (see equation (5.13) on page 137):

$$\delta \geq \max\{D(\text{OR-Tree}(e_{UQ})), 2 \cdot D_{AND}\} + 3 \cdot D_{AND}.$$

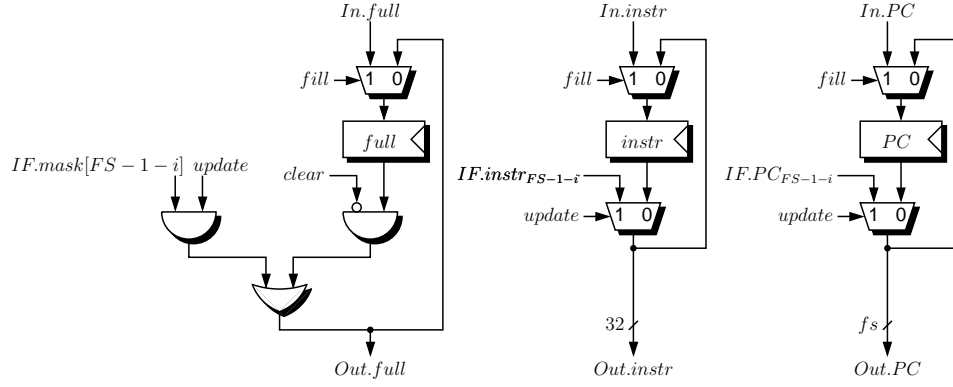


Figure 7.2: Optimized IFQ entry

Since this does not hold for $\delta = 4$ the optimized completion cannot be used for $\delta = 4$. The signals $sent_\star$ bound the stage depth δ by (see equation (5.14) on page 137)

$$\delta \geq D(\text{FFO}(e_{UQ} + 1)) + D_{AND} + D_{MUX}$$

which holds true for $e_{UQ} = 1$ and $\delta = 4$.

For the read queue it must hold (see equation (5.17) on page 141):

$$\delta \geq \begin{cases} 3 \cdot D_{AND} & \text{if } e_{RQ} = 1 \\ \max\{D_{AND}, D(\text{FFO}(e_{RQ}))\} + 2 \cdot D_{AND} + D_{MUX} & \text{if } e_{RQ} > 1 \end{cases}$$

Thus, the number of read queue entries e_{RQ} must also be one for $\delta = 4$.

The instruction fetch queue requires that the stage depth fulfills the equation (6.1) on page 161

$$\delta \geq 2 \cdot D_{AND} + D_{OR} + D_{MUX}$$

which does not hold for $\delta = 4$. In order to be able to build the IFQ for $\delta = 4$ the following modifications have to be made: the AND-ing of the full bit and the mask is done in the instruction fetch unit. This can be done without increasing the delay, but upon activation of the clear signal all mask bits have to be reset. Then the circuit depicted in figure 7.2 can be used for the entries of the instruction fetch queue. Note that in this optimized version the entries might be filled even if the clear signal is active. Thus, the clear signal for the IFQ must be active at least two cycles.

Using the modifications described in this section it is possible to build the $\text{DLX}_{\pi+}$ with a stage depth δ of 4. However, equation (4.14) on page 59 bounds the number of reservation station entries for all reservation stations to one. Note that the reservation stations stall output is active even if an instruction is currently dispatched, since otherwise the delay of the stall output would get too large. Thus, instructions can be issued to reservation stations with one entry only every other cycle.

Thus, in order to be able to issue one instruction of a type in every cycle the $\text{DLX}_{\pi+}$ needs two functional units of the type. Since the maximum number of functional units of a type is two due to equation (4.2) on page 43, at most two instructions

of a type can wait for their operands. Since for such a deep pipeline it is very likely that the operands of an instruction are not valid at the time the instruction is issued, this is assumed to have a large performance impact.

Also, the optimization of the update queue from figure 5.21 on page 137 cannot be done (see equation 5.16 on page 139). Thus, despite the optimized completion, stores can only be completed if the stall from the memory unit to the read queue of the data cache is inactive.

For these reasons and the large increase in the number of cycles needed it is very unlikely that the performance of the $DLX_{\pi+}$ regarding a reasonable benchmark can be improved by reducing the stage depth from 5 to 4. Therefore, in this thesis only stage depths $\delta \geq 5$ were discussed in detail.

This thesis does not provide a formal proof that it is impossible to build the $DLX_{\pi+}$ with a stage depth $\delta = 3$ without sacrificing fundamental aspects of the $DLX_{\pi+}$, e.g., the out-of-order execution, the precise interrupts, or the possible best case CPI (cycles-per-instruction) of 1. However, the author thinks that it is at least very unlikely that such a $DLX_{\pi+}$ is possible, since even after the described enhancements most equations bound the stage depth to be at least 4. In particular the bound given by the loop for the data of the forwarding circuit with stalling (see equation (2.12) on page 23) that bounds delta to

$$\delta \geq 2 \cdot D_{MUX}$$

is assumed to be hard to improve.

7.2 Gate Model

The gate model used in this thesis is rather simple, since it does not take fanout or wire delays into consideration. Note that some of the path that bound the stage depth δ have large fanout, e.g., the signal *ROBhead.valid* which stalls the whole ROB read access in the retiring context. Due to the reduction of the device sizes in integrated circuits the delay of wires will have an increasing influence on the combinational delay of signals and therefore should not be neglected [HHM99]. This can also affect some of the critical paths (in particular stall signals) of the $DLX_{\pi+}$. Thus, the gate model used in this thesis could be too optimistic regarding delay.

On the other hand the gate model only allows two-input gates and two-port muxes. Thus, using the more complex gates available in real designs like multi-input NANDs and NORs or wide transmission-gate-muxes [WH04] the delay of the critical path could be reduced. This could make up for the too optimistic estimations of the gate model due to the neglected fanout and wire delays.

Even if one assumes that the gate model is not very realistic, the degree of pipelining presented in this thesis is still very high in comparison to previous work. For example in the deeply pipelined Pentium 4 processor a 16 bit addition (which has 12 combinational gate delays in our model) can be in less than half a cycle [HSU⁺01]. Thus, even if the used gate model is off by a factor of two, the amount of useful work that can be done in a cycle of the Pentium 4 is still more than three times more than the 4 gate delays of the $DLX_{\pi+}$ with minimum cycle time.

| Variable | Description |
|-------------------------|--|
| $\tau = \delta + 5$ | cycle time of the $DLX_{\pi+}$ |
| $L_{ROB} = 2^{l_{ROB}}$ | # lines of the reorder buffer |
| n_i | # FU's of type i ($0 \leq i \leq 6$) |
| e_{RS_i} | # entries of the RSs of type i ($0 \leq i \leq 6$) |
| $L_{DC} = 2^{l_{DC}}$ | # lines of the data cache |
| $S_{DC} = 2^{s_{DC}}$ | # bytes of the cache-lines of the data cache |
| $K_{DC} = 2^{k_{DC}}$ | associativity of the data cache |
| e_{UQ} | # entries of the update queue |
| e_{RQ} | # entries of the read queue |
| $L_{IC} = 2^{l_{IC}}$ | # lines of the instruction cache |
| $S_{IC} = 2^{s_{IC}}$ | # bytes of the cache-lines of the instr. cache |
| $K_{IC} = 2^{k_{IC}}$ | associativity of the instruction cache |
| $L_{BTB} = 2^{l_{BTB}}$ | # lines of the branch target buffer |
| $K_{BTB} = 2^{k_{BTB}}$ | associativity of the branch target buffer |
| $FS = 2^{fs}$ | # instructions fetched per $IFCLK$ cycle |

Table 7.1: Parameters of the $DLX_{\pi+}$

The author thinks that even for a more realistic model (e.g., the logical effort model [SSH99]) it would still be possible to allow a similarly extensive super-pipelining. In addition to the techniques discussed in this thesis it would probably be necessary to further decrease the delay of stall signals, e.g., by pipelining the full bits of an instruction one cycle ahead of the actual data and thus computing the stall signal one cycle ahead. Also the functional units (except the memory unit) would not need stall signals if the number of CDBs were equal to the number of functional units and instructions could not “collide” inside the functional units. Collisions could be prevented, e.g., by handling divisions in software and therefore resolving loops, or by adapting the dispatch logic.

7.3 Overall Cost and Delay

In this section the variables describing the pipelining of the $DLX_{\pi+}$ (e.g., the number of cycles needed for decoding) in dependence of the parameters (e.g., cycle time or ROB size) are presented. The parameters of the $DLX_{\pi+}$ are listed in table 7.1. The variables describing the behavior of the $DLX_{\pi+}$ are listed in table 7.2.

Using the formulas presented in this thesis it is straightforward to write a small program that computes the variables and the cost of the $DLX_{\pi+}$ in dependence of the parameters.¹ In appendix E the values of the variables for some combinations of the parameters are described.

Table 7.3 compares the $DLX_{\pi+}$ with the out-of-order DLX presented in [Krö99] which is in the following called DLX_K . In order to match the DLX_K , the following parameters are chosen for the $DLX_{\pi+}$:

¹The program can be found at
<http://www-wjp.cs.uni-sb.de/leute/private/homepages/jochen/DLX+.tgz>.

| Variable | Description |
|--------------|--|
| m | multiplier for $IFCLK$ |
| c_{IF} | # $IFCLK$ cycles of the instruction fetch |
| c_{D1} | # cycles needed for the decode sub-phase $D1$ |
| c_{D2} | # cycles needed for the ROB access in the decode sub-phase $D2$ |
| c_I | # cycles needed for issuing |
| c_T | # cycles (needed if $c_I > 1$) |
| c_{U2DM} | minimum # cycles from the update of an instruction in the memory RS to the dispatch of the instruction |
| c_{U2DI} | minimum # cycles from the update of an instruction in an integer or BCU RS to the dispatch of the instruction |
| c_{U2DF} | minimum # cycles from the update of an instruction in an FP RS to the dispatch of the instruction |
| c_{U2DB} | minimum # cycles from the update of an instruction in an branch checking RS to the dispatch of the instruction |
| c_{AT} | # cycles needed for the arbiter tree in the complete phase |
| c_C | # cycles needed for the complete phase |
| c_{C2R} | minimum # cycles from the completion to the retiring of an instruction |
| c_{Ret2} | # cycles needed for the retire sub-phase $Ret2$ |
| c_{Sh4S} | # cycles needed for the shift for store circuit |
| c_{HC} | # cycles needed for the hit computation |
| c_{M2H} | # cycles the result bus must be delayed |
| c_{M2W} | minimum # cycles between the returning of a memory request and the update of the cache core by a store in the update queue |
| c_{M2R} | minimum # cycles between the returning of a memory request and the update of the cache core by a load in the update queue |
| c_{M2Q} | minimum # cycles between the returning of a memory request and the returning of a load by the read queue |
| c_{Sh4L} | # cycles needed for the shift for load circuit |
| c_{IAlu} | # cycles needed for the integer ALU |
| c_{IMul1} | # cycles needed for the first part of the integer multiplicative unit |
| c_{IMul2} | # cycles needed for the second part of the integer multiplicative unit |
| c_{IMul3} | # cycles needed for the third part of the integer multiplicative unit |
| c_{FPAdd} | # cycles needed for the FP additive unit |
| c_{FPMul1} | # cycles needed for the first part of the FP multiplicative unit |
| c_{FPMul2} | # cycles needed for the second part of the FP multiplicative unit |
| c_{FPMul3} | # cycles needed for the third part of the FP multiplicative unit |
| c_{FPMul4} | # cycles needed for the fourth part of the FP multiplicative unit |
| c_{FPMisc} | # cycles needed for the FP miscellaneous unit |
| c_{BCU} | # cycles needed for the branch checking unit |

Table 7.2: Variables describing the behavior of the $DLX_{\pi+}$

- The ROB has 16 entries.
- Every functional unit type is instantiated only once. In the DLX_K the reservation stations of the floating point units have two entries, all other reservation stations have four entries. The reservation stations of the DLX_K can issue an instruction into a reservation entry in the same cycle an instruction is dispatched from the reservation station, even if the reservation station is full. This is not possible in the $DLX_{\pi+}$, see section 4.2.2. Therefore, the number of entries is increased by one for the $DLX_{\pi+}$.
- The DLX_K uses a common 16KB direct-mapped cache for data and instructions. In order to match the size, the $DLX_{\pi+}$ uses two 8KB direct-mapped caches for the instruction fetch and the memory unit.
- The other parameters have no counterpart in the DLX_K and are chosen as follows: update queue and read queue have 4 entries, the BTB has 256 entries and is 4-way set associative. The instruction fetch unit fetches 8 instructions per $IFCLK$ -cycle.

The cycle time of the DLX_K is 106 [Krö99]. The cycle time τ of the $DLX_{\pi+}$ must be at least 12 ($\delta = 7$) in order to allow five reservation station entries (see section 4.2.3). Thus, for cycle times $\tau \in \{10, 11\}$ the number of reservation stations is set to at most 2 respectively 4. Note that in table 7.3 the columns for those values of the cycle time τ have been omitted which would have the same entries as the column for the next smaller τ .

Only the variables m and c_{IF} change their value for cycle times greater the 37. At a cycle time $\tau = 50$ the multiplier m can be set to one, but then the instruction cache access must be pipelined, i.e., c_{IF} becomes 2. If $\tau = 68$ both values m and c_{IF} can be one. Since the instruction fetch unit delivers up to 8 instruction per instruction fetch cycle, is not assumed to be performance critical if the instruction fetch takes two cycles. Therefore, in the following only cycles times of 37 and below are investigated for the $DLX_{\pi+}$.

Note that the DLX_K does not divide the decode phase into the sub-phases $D1$ and $D2$. Therefore, the number of cycles needed for issuing and for the ROB access in the decode sub-phase $D2$ are set to zero for the DLX_K . For cycles times of 30 and above, the value c_{IALU} of the $DLX_{\pi+}$ is zero. This indicates that the integer ALU does not contain any registers. Thus, the instruction can directly proceed from the reservation station through the integer ALU to the CDB register. In the the DLX_K all functional units must have a register, but the completion phase has no register, which has the same effect. In order to simplify the comparison the variables c_{IALU} and c_C of the DLX_K are set to the same value as for the $DLX_{\pi+}$ with a cycle time of 30.

The $DLX_{\pi+}$ with a cycle time of 37 needs in the best case 5 cycles to process an integer ALU instruction (instruction fetch not counted): 1 cycle decode, 1 cycle issue, 1 cycle dispatch + execution, 1 cycle completion, and 1 cycle retiring. This is one cycle more than needed for the DLX_K which does decoding and issuing in the same cycle. However, the cycle time of the DLX_K is roughly 3 times as large as the cycle time of this variant of the $DLX_{\pi+}$. Thus, even without super-pipelining the $DLX_{\pi+}$ is assumed to have a much better performance than the DLX_K .

| τ | DLX $_{\pi+}$ | | | | | | | | | | | | | | | | | | | | DLX $_K$ |
|------------|---------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----|----------|
| | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 20 | 21 | 23 | 25 | 30 | 34 | 35 | 37 | 50 | 68 | 106 | |
| m | 5 | 5 | 5 | 4 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | |
| c_{IF} | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 1 | |
| c_{D1} | 7 | 6 | 5 | 4 | 4 | 4 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | |
| c_{D2} | 4 | 4 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | |
| c_I | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | |
| c_{U2DM} | 3 | 4 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| c_{U2DI} | 4 | 4 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| c_{U2DF} | 5 | 4 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| c_{U2DB} | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| c_C | 3 | 3 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| c_{C2R} | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| c_{Ret2} | 7 | 6 | 5 | 5 | 4 | 4 | 4 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | |
| c_{IALU} | 4 | 3 | 3 | 2 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Table 7.3: Comparison of selected variables of the DLX $_{\pi+}$ and DLX $_K$

| τ | DLX $_{\pi+}$ | | | | | | | DLX $_K$ |
|----------|---------------|---------|---------|---------|---------|--------|--------|-----------------|
| | 10 | 11 | 12 | 15 | 18 | 25 | 37 | 106 |
| IF/BCU | 344895 | 342703 | 339568 | 337376 | 335337 | 334241 | 291155 | 189741 |
| Decode | 22637 | 21125 | 17764 | 9001 | 7489 | 5977 | 4465 | 3970 |
| Dispatch | 216404 | 314301 | 272798 | 193953 | 169423 | 74372 | 74372 | 43679 |
| Complete | 11776 | 12684 | 7268 | 7268 | 7268 | 7268 | 7268 | 196 |
| Ret./ROB | 49464 | 43650 | 35197 | 28106 | 26648 | 16822 | 15364 | 19807 |
| RF | 23433 | 20055 | 18103 | 14722 | 14184 | 11856 | 11670 | 19545 |
| PT | 29690 | 25117 | 19802 | 12320 | 12320 | 7083 | 4966 | 15574 |
| Mem | 1202498 | 973211 | 904068 | 601757 | 507442 | 329466 | 277848 | 225336 |
| IALU | 4502 | 4038 | 4038 | 3110 | 3110 | 3110 | 2646 | 3693 |
| IMul | 16813 | 15445 | 14533 | 14077 | 12709 | 12709 | 12253 | na ² |
| FPAdd | 43447 | 38967 | 36279 | 32695 | 30903 | 27319 | 25527 | 23735 |
| FPMul | 75333 | 69957 | 65477 | 59205 | 56517 | 52933 | 50245 | 47557 |
| FPMisc | 19510 | 18614 | 17718 | 16822 | 16822 | 15926 | 15926 | 18135 |
| Total | 2060402 | 1899867 | 1752613 | 1330412 | 1200172 | 899082 | 793705 | 610968 |

Table 7.4: Comparison of the costs of the DLX $_{\pi+}$ and the DLX $_K$

The critical path of the DLX $_K$ starts at the instruction register, goes through the de-code phase and finally updates the PC register which determines the address of the next instruction. The last part of this path has been removed for the DLX $_{\pi+}$ by using branch prediction. Then by splitting the decode phase into two sub-phases and improving the circuits of the decode phase, the cycle time could be dramatically improved without increasing the number of stages significantly.

Table 7.4 list the cost of the DLX $_{\pi+}$ and the DLX $_K$ for some selected cycle times

²The DLX $_K$ does not support integer multiplications and divisions.

| τ | 10 | 11 | 12 | 15 | 18 | 25 | 37 |
|----------|--------|--------|--------|--------|--------|--------|--------|
| IF/BCU | 118.5% | 117.7% | 116.6% | 115.9% | 115.2% | 114.8% | 100.0% |
| Decode | 507.0% | 473.1% | 397.8% | 201.6% | 167.7% | 133.9% | 100.0% |
| Dispatch | 291.0% | 422.6% | 366.8% | 260.8% | 227.8% | 100.0% | 100.0% |
| Complete | 162.0% | 174.5% | 100.0% | 100.0% | 100.0% | 100.0% | 100.0% |
| Ret./ROB | 321.9% | 284.1% | 229.1% | 182.9% | 173.4% | 109.5% | 100.0% |
| RF | 200.8% | 171.9% | 155.1% | 126.2% | 121.5% | 101.6% | 100.0% |
| PT | 597.9% | 505.8% | 398.8% | 248.1% | 248.1% | 142.6% | 100.0% |
| Mem | 432.8% | 350.3% | 325.4% | 216.6% | 182.6% | 118.6% | 100.0% |
| IALU | 170.1% | 152.6% | 152.6% | 117.5% | 117.5% | 117.5% | 100.0% |
| IMul | 137.2% | 126.1% | 118.6% | 114.9% | 103.7% | 103.7% | 100.0% |
| FPAdd | 170.2% | 152.7% | 142.1% | 128.1% | 121.1% | 107.0% | 100.0% |
| FPMul | 149.9% | 139.2% | 130.3% | 117.8% | 112.5% | 105.3% | 100.0% |
| FPMisc | 122.5% | 116.9% | 111.3% | 105.6% | 105.6% | 100.0% | 100.0% |
| Total | 259.6% | 239.4% | 220.8% | 167.6% | 151.2% | 113.3% | 100.0% |

Table 7.5: Relative increase of the cost of the $DLX_{\pi+}$

from table 7.3. The cost of the cache of the DLX_K were evenly distributed to the instruction fetch and the memory unit. The higher cost of the $DLX_{\pi+}$ with cycle time 37 compared to the DLX_K is mainly due to the branch prediction and the non-blocking cache, which increase the cost of the instruction fetch respectively the memory unit.

Table 7.5 lists the relative increase of the cost for the $DLX_{\pi+}$ if the cycle time is reduced (normalized to the $DLX_{\pi+}$ with a cycle time of 37). The increase of the cost is only due to the additional forwarding circuits and registers, the number of gates for the actual computations is the same. The number of registers that has to be added due to pipelining is reciprocal to the stage depth. Note that forwarding circuits and queues use two-dimensional pipelining, i.e., the number of registers increases quadratically if the stage depth is reduced.

Queues are used in the memory unit and the dispatch phase. Extensive forwarding is used in the ROB and the producer table environment. Therefore, these parts have a large relative increase. The cost of the dispatch get smaller again for $\tau \in \{10, 11\}$ since the number of reservation station entries has to be reduced for these cycle times. The cost of the decode phase increases drastically if the issue circuit has to be pipelined. The cost of the instruction fetch is not sensitive to the decrease of the cycle time as the multiplier m for the instruction fetch clock $IFCLK$ increases if the cycle time decreases. Since the total cost of the instruction fetch is high this reduces the relative increase of the $DLX_{\pi+}$.

7.4 Related Work

Super-pipelining is a well-known technique to improve processor performance. It is used in many commercial processors, e.g., in the succeeding generation of the MIPS R3000 processor on which the $DLX_{\pi+}$ ISA is based on [BLM91]. Other work combines super-pipelining with other techniques such as super-scalar designs [JCL94], multi-threading [GV95], or both [GV96]. However these studies use rather moderate

super-pipelining which at least allows the computation of a 16-bit addition (which has 12 combinational gate delays in our model) in a single cycle [GV95]. Note that in the deeply pipelined Pentium 4 a 16-bit addition can be computed even with double the core frequency [HSU⁺01].

Extreme super-pipelining as done in this thesis is considered in previous work that studies the optimum pipeline depth of a processor [KS86][HJF⁺02][SC02][HP02]. Usually these studies are based on an existing processor. They compute the combinational work to be done per instruction from the cycles needed and the stage depth of the processor. The cycles needed for other frequencies is then simply computed by dividing the combinational work by the corresponding stage depth. They do not take into account that the combinational work may depend on the stage depth, e.g., if forwarding of RAM ports becomes necessary (see section 2.6.1). The optimal stage depth is obtained by simulating a benchmark suite for different processor frequencies. None of the studies details the effects on the logic if the cycle time is reduced or assumes a lower bound for the stage depth. We now discuss the cited work in some more detail.

An initial study on the optimum pipeline depth of a processor was done by Kunkel and Smith [KS86]. Their work is based on a CRAY-1S supercomputer build from discrete ECL gates. Kunkel and Smith assume that interlocking (i.e., stalling) must be computed in one cycle and that central RAM blocks must be accessed in one cycle. These bounds can be overcome using the techniques used in this thesis. Limitations to the minimum stage depth are noted in [KS86] but are later ignored in the simulations. They conclude that the maximum performance is achieved with a stage depth of 8 ECL gate levels for scalar code (4 levels for vector code).

Hrishikesh et. al. [HJF⁺02] studied the optimum stage depth of an Alpha 21264 with large register files and large cache. They assume that the instruction wakeup and instruction select logic are critical circuits that bound the stage depth. In order to allow lower stage depths Hrishikesh et. al. propose to pipeline these circuits by dividing the issue window in multiple stages. They do not discuss the consequences of this pipelining to the correctness of the logic, e.g., if an instruction moves from one stage to the next. Using these pipelined circuits and assuming that all other circuits can be perfectly pipelined into arbitrary stages Hrishikesh et. al. conclude the optimum stage depth to be 6 FO4³ for integer benchmarks and 4 FO4 for floating point benchmarks.

Sprangle and Carmean [SC02] measured the performance loss s for some critical loops (ALU, branch prediction, cache accesses) if the loops are increased by one cycle. They model the performance loss for a loop with m additional cycles to be $(1 - s)^m$. The total performance is computed as product of the losses for the loops. This model matches their simulations. They derive that the performance of a Pentium 4 processor can be improved by 35 to 90% through implementing deeper pipelines and larger caches. Sprangle and Carmean assume that all circuits can be pipelined arbitrarily. They address the problem of forwarding in order to pipeline RAM accesses but do not handle pipelining of the forwarding itself and do not include forwarding in their simulations.

Hartstein and Puzak [HP02] developed an analytical formula for the performance of a S/390 processor in dependence of the stage depth. The optimum stage depth is found by equating the derivation function to zero. Yet the formula uses some processor

³Fan-out-of-four (FO4) is defined as the delay of an inverter that drives four inverters of the same size.

and benchmark dependent constants which are hard to determine. Also the formula assumes that processor logic can be uniformly divided into an arbitrary number of stages.

Recent studies [HP03] [SBG⁺02] also take power consumption into account. Since low stage depths dramatically increase the power consumption due to the large amount of necessary latches this generally leads to a larger optimum pipeline depth. Since the power consumption depends in first order on the cost of processor, this can be included in our model by not just optimizing the performance of the processor but a quality metric

$$Quality = \frac{1}{Performance^{1-q} \cdot Cost^q}$$

for a $q \in [0; 1]$ as proposed by Grün [Grü94].

To the author's best knowledge the theoretical limits of super-pipelining and the minimum stage depth as presented in this thesis have not yet been studied. Also the consequences to the logic if even the forwarding circuits are pipelined has not yet been discussed.

Chapter 8

Summary

In this thesis the basic techniques needed for super-pipelining of processors were described. These techniques comprise the insertion of buffer circuits to split the stall trees and the pipelining of RAM blocks. In order to pipeline RAM blocks it is necessary to forward the data written by succeeding writes to ongoing read accesses. The forwarding circuits presented in this thesis even allow to reduce the cycle time below the time needed for forwarding data by pipelining the forwarding circuit itself, which introduces two-dimensional pipelining.

Using these techniques the $DLX_{\pi+}$, an out-of-order processor with multiple variable parameters including the cycle time was presented. The cycle time of the $DLX_{\pi+}$ can be reduced to down to five gate delays, with restrictions even down to four gate delays. Formulas were developed that compute the cost and the number of pipeline stages based on the cycle time. Correctness proofs were given for the parts that differ significantly from the DLX presented in [Krö99]. In particular for every RAM block it was proven that the forwarding circuits deliver the correct data needed for the overall correctness of the Tomasulo algorithm.

8.1 Future Work

Using the formulas presented in this thesis it is possible to compute the variables describing the pipelining of the $DLX_{\pi+}$ in dependence of the parameters. The next step would be to write a simulator for the $DLX_{\pi+}$. This simulator could use instruction traces of a benchmark suite in order to compute the average time per instruction (TPI) of the $DLX_{\pi+}$ configured to different cycle times, ROB sizes, etc. for this benchmark suite. This would allow to examine the parameters of the $DLX_{\pi+}$ that deliver the best TPI values. In particular the optimal cycle time for the $DLX_{\pi+}$ could be determined.

Apart from super-pipelining the number of active instructions in a processor can be increased by issuing multiple instructions in one cycle (super-scalar processors). The $DLX_{\pi+}$ only supports issuing of one instruction per cycle. A super-scalar DLX is presented in [Hil00], methods to reduce the cycles time of the central circuits of super-scalar processors can be found, e.g., in [PJS97]. An expansion of the $DLX_{\pi+}$ could combine multi-issuing and super-pipelining to investigate the advantages of the different approaches.

Modern processors are able to execute multiple threads at the same time. This al-

lows for better utilization of the resources. Using multi-threading in a super-pipelined processor could decrease the optimum cycle time since useful work can be done while an instruction waits for data.

The $DLX_{\pi+}$ uses a rather simple branch prediction scheme. The performance impact of the branch prediction increases if the cycle time is reduced, since this increases the number of cycles needed for the roll-back of a mispredicted branch. Hence, increasing the hit-rate of the branch prediction would be worthwhile. Also the roll-back could be improved, such that the mispredicted instruction do not need to retire before the succeeding instructions from the corrected branch target can be decoded.

The cost and delay calculations of the $DLX_{\pi+}$ are based on a rather simple gate model that does not take fanout and wiring into account. Since the stall signals of the $DLX_{\pi+}$ have a large fanout, a gate model that includes fanout, e.g. the logical effort model [SSH99], could have significant impact on the bounds of the cycle time. In order to investigate the impact, the delay calculations of the $DLX_{\pi+}$ could be adopted for the logical effort model.

Appendix A

Instruction set architecture

A.1 Instructions

In this section the instructions which are supported by the processor are summarized. All instructions which do not handle special registers or the program counter, are directly taken from the MIPS 32000 instruction set [KH92]. Since the processor does not support delayed branch, the control flow change instructions are altered accordingly. Special register and interrupt handling are based on the DLX implementation by Müller and Paul [MP00].

The instructions of the processor are presented in the tables A.1 to A.7, ordered by the type of functional unit they use. If the bits [31 : 26] of the instruction indicate an floating point instruction, but the rest of the instruction matches none of the following opcodes, the *uFOP* (unimplemented floating point operation) interrupt is raised. Otherwise if the instruction matches none of the following opcodes, the *ill* (illegal instruction) interrupt is raised. In this case the instruction is not sent to an functional unit but the valid bit of the ROB entry is set.

The instructions have up to four parameters: the destination register address D , the two operand addresses OP_1 and OP_2 , and an immediate constant imm . For the description of the instructions, the following abbreviations are used for the register files:

$$\begin{array}{lll} GD := GPR[D] & FD := FPR[D] & SD := SPR[D] \\ GOP_1 := GPR[OP_1] & FOP_1 := FPR[OP_1] & SOP_1 := SPR[OP_1] \\ GOP_2 := GPR[OP_2] & FOP_2 := FPR[OP_2] & SOP_2 := SPR[OP_2] \\ Gi := GPR[i] & & Si := SPR[i] \end{array}$$

The following abbreviation is used to describe a memory location of variable width (if the memory is seen as an one-dimensional array):

$$M(addr, d) := M[8 \cdot (addr + d) - 1 : 8 \cdot addr]$$

The memory is usually addressed by the sum of the operand 1 and the immediate constant. This sum is abbreviated by *ea*. For the description of the instruction *lwl*, *lwr*, *swl*, and *swr* (load / store word left / right), the base address *ba* and the offset

| IR[31:26] | Instr. | Group | Effect |
|-----------|--------|---------|---|
| 100000 | lb | Load | $[GD] = [M(ea, 1)]$ |
| 100001 | lh | | $[GD] = [M(ea, 2)]$ |
| 100011 | lw | | $[GD] = [M(ea, 4)]$ |
| 100100 | lbu | | $\langle GD \rangle = \langle M(ea, 1) \rangle$ |
| 100101 | lhu | | $\langle GD \rangle = \langle M(ea, 2) \rangle$ |
| 100010 | lwl | LoadLR | $GD = M(ba, oa + 1), GD[23 - 8 \cdot oa : 0]$ |
| 100110 | lwr | | $GD = GD[31 : 8 \cdot (4 - oa)], M(ea, 4 - oa)$ |
| 110001 | lwc1 | LoadFP | $[FD] = [M(ea, 4)]$ |
| 101000 | sb | Store | $M(ea, 1) = GOP_2[7 : 0]$ |
| 101001 | sh | | $M(ea, 2) = GOP_2[15 : 0]$ |
| 101011 | sw | | $M(ea, 4) = GOP_2$ |
| 101010 | swl | StoreLR | $M(ba, oa + 1) = GOP_2[31 : 8 \cdot (3 - oa)]$ |
| 101110 | swr | | $M(ea, 4 - oa) = GOP_2[8 \cdot (4 - oa) - 1 : 0]$ |
| 111001 | swc1 | StoreFP | $M(ea, 4) = FOP_2$ |

Table A.1: Memory instructions

address oa are used:

$$ea := [GOP_1] + [imm]$$

$$ba := \lfloor ea/4 \rfloor$$

$$oa := ea \bmod 4$$

A.2 Encoding

The table A.8 shows the encoding of the instructions parameters D , OP_1 , OP_2 , and imm . The parameters D , OP_1 , and OP_2 have always width 5 bit. The width of the immediate constant imm depends on the width of the imm field in the encoding.

| IR[31:26] | Instr. | Group | Effect |
|-----------|--------|-------|---|
| 001000 | addi | AluI | $[GD] = [GOP_1] + [imm]$ |
| 001001 | addiu | | $\langle GD \rangle = \langle GOP_1 \rangle + \langle imm \rangle$ |
| 001010 | slti | | $GD = ([GOP_1] < [imm]) ? 0^{31}1 : 0^{32}$ |
| 001011 | sltiu | | $GD = (\langle GOP_1 \rangle < \langle imm \rangle) ? 0^{31}1 : 0^{32}$ |
| 001100 | andi | | $GD = GOP_1 \wedge (0^{16}, imm)$ |
| 001101 | ori | | $GD = GOP_1 \vee (0^{16}, imm)$ |
| 001110 | xori | | $GD = GOP_1 \oplus (0^{16}, imm)$ |
| 001111 | lui | | $GD = (imm, 0^{16})$ |
| 111111 | trap | Trap | $trap = 1, \quad eData = [imm]$ |

| $IR[31 : 26] = 000000$ | | | |
|------------------------|--------|--------|---|
| IR[5:0] | Instr. | Group | Effect |
| 000000 | sll | ShiftI | $GD = GOP_2 \ll \langle imm \rangle$ |
| 000010 | srl | | $GD = GOP_2 \gg \langle imm \rangle$ |
| 000011 | sra | | $GD = GOP_2 \gg \langle imm \rangle$ (arith.) |
| 000100 | sllv | Shift | $GD = GOP_2 \ll \langle GOP_1[4 : 0] \rangle$ |
| 000110 | srlv | | $GD = GOP_2 \gg \langle GOP_1[4 : 0] \rangle$ |
| 000111 | srav | | $GD = GOP_2 \gg \langle GOP_1[4 : 0] \rangle$ (arith.) |
| 100000 | add | Alu | $[GD] = [GOP_1] + [GOP_2]$ |
| 100001 | addu | | $\langle GD \rangle = \langle GOP_1 \rangle + \langle GOP_2 \rangle$ |
| 100010 | sub | | $[GD] = [GOP_1] - [GOP_2]$ |
| 100011 | subu | | $\langle GD \rangle = \langle GOP_1 \rangle - \langle GOP_2 \rangle$ |
| 100100 | and | | $GD = GOP_1 \wedge GOP_2$ |
| 100101 | or | | $GD = GOP_1 \vee GOP_2$ |
| 100110 | xor | | $GD = GOP_1 \oplus GOP_2$ |
| 100111 | nor | | $GD = \overline{GOP_1 \vee GOP_2}$ |
| 101010 | slt | | $GD = ([GOP_1] < [GOP_2]) ? 0^{31}1 : 0^{32}$ |
| 101011 | sltu | | $GD = (\langle GOP_1 \rangle < \langle GOP_2 \rangle) ? 0^{31}1 : 0^{32}$ |

| $IR[31 : 26] = 010001$ | | | |
|------------------------|---------|--------|--------------|
| IR[25:21] | Instr. | Group | Effect |
| 00000 | movef2i | Move2I | $GD = FOP_2$ |
| 00010 | moves2i | | $GD = SOP_2$ |
| 00100 | movei2f | MoveI2 | $FD = GOP_2$ |
| 00110 | movei2s | | $SD = GOP_2$ |

Table A.2: Integer ALU instructions

| $IR[31 : 26] = 000000$ | | | | |
|------------------------|---------------|-------|--|--|
| IR[5:0] | Instr. | Group | Effect | |
| 011000 011001 | mult multu | Mult | $[S9, S8] = [GOP_1] * [GOP_2]$ $\langle S9, S8 \rangle = \langle GOP_1 \rangle * \langle GOP_2 \rangle$ | |
| 011010 011011 | div divu | Div | $[S8] = [GOP_1] / [GOP_2]$ $[S9] = [GOP_1] \bmod [GOP_2]$ $\langle S8 \rangle = \langle GOP_1 \rangle / \langle GOP_2 \rangle$ $\langle S9 \rangle = \langle GOP_1 \rangle \bmod \langle GOP_2 \rangle$ | |

Table A.3: Integer multiplicative instructions

| $IR[31 : 26] = 010001$ | | | | |
|------------------------|--------------------------------------|--------------------------------------|-------|--|
| IR[21] | IR[5:0] | Instr. | Group | Effect |
| 0 0 1 1 | 000000 000001 000000 000001 | fadd.s fsub.s fadd.d fsub.d | FAdd | $\llbracket FD \rrbracket = \llbracket FOP_1 \rrbracket + \llbracket FOP_2 \rrbracket$ $\llbracket FD \rrbracket = \llbracket FOP_1 \rrbracket - \llbracket FOP_2 \rrbracket$ $\llbracket FD^+ \rrbracket = \llbracket FOP_1^+ \rrbracket + \llbracket FOP_2^+ \rrbracket$ $\llbracket FD^+ \rrbracket = \llbracket FOP_1^+ \rrbracket - \llbracket FOP_2^+ \rrbracket$ |

Table A.4: Floating point additive instructions

| $IR[31 : 26] = 010001$ | | | | |
|------------------------|--------------------------------------|--------------------------------------|-------|--|
| IR[21] | IR[5:0] | Instr. | Group | Effect |
| 0 0 1 1 | 000010 000011 000010 000011 | fmul.s fdiv.s fmul.d fdiv.d | FMul | $\llbracket FD \rrbracket = \llbracket FOP_1 \rrbracket * \llbracket FOP_2 \rrbracket$ $\llbracket FD \rrbracket = \llbracket FOP_1 \rrbracket / \llbracket FOP_2 \rrbracket$ $\llbracket FD^+ \rrbracket = \llbracket FOP_1^+ \rrbracket * \llbracket FOP_2^+ \rrbracket$ $\llbracket FD^+ \rrbracket = \llbracket FOP_1^+ \rrbracket / \llbracket FOP_2^+ \rrbracket$ |

Table A.5: Floating point multiplicative instructions

| $IR[31 : 26] = 010001$ | | | | |
|--|--|--|--------|--|
| IR[23:21] | IR[5:0] | Instr. | Group | Effect |
| 000 001 | 11c[3:0] 11c[3:0] | fcomp.s fcomp.d | FComp | $S7 = (\llbracket FOP_1 \rrbracket op \llbracket FOP_2 \rrbracket) ? 1 : 0$ $S7 = (\llbracket FOP_1^+ \rrbracket op \llbracket FOP_2^+ \rrbracket) ? 1 : 0$ |
| 000 000 000 001 001 001 | 000101 000110 000111 000101 000110 000111 | fabs.s fmov.s fneg.s fabs.d fmov.d fneg.d | FMisc | $\llbracket FD \rrbracket = \llbracket FOP_1 \rrbracket $ $\llbracket FD \rrbracket = \llbracket FOP_1 \rrbracket$ $\llbracket FD \rrbracket = -\llbracket FOP_1 \rrbracket$ $\llbracket FD^+ \rrbracket = \llbracket FOP_1^+ \rrbracket $ $\llbracket FD^+ \rrbracket = \llbracket FOP_1^+ \rrbracket$ $\llbracket FD^+ \rrbracket = -\llbracket FOP_1^+ \rrbracket$ |
| 001 100 | 100000 100000 | fcvt.s.d fcvt.s.w | FCvt.S | $\llbracket FD \rrbracket = \llbracket FOP_1^+ \rrbracket$ $\llbracket FD \rrbracket = \llbracket FOP_1 \rrbracket$ |
| 000 001 100 | 100001 100001 100001 | fcvt.d.s fcvt.d.d fcvt.d.w | FCvt.D | $\llbracket FD^+ \rrbracket = \llbracket FOP_1 \rrbracket$ $\llbracket FD^+ \rrbracket = \llbracket FOP_1^+ \rrbracket$ $\llbracket FD^+ \rrbracket = \llbracket FOP_1 \rrbracket$ |
| 000 001 | 100100 100100 | fcvt.w.s fcvt.w.d | FCvt.W | $\llbracket FD \rrbracket = \llbracket FOP_1 \rrbracket$ $\llbracket FD \rrbracket = \llbracket FOP_1^+ \rrbracket$ |

Table A.6: Floating miscellaneous instructions

| IR[31:26] | Instr. | Group | Effect |
|-----------|--------|---------|--|
| 000100 | beq | Branch | $[PC] + = ([GOP_1] = [GOP_2])? [imm] : 4$ |
| 000101 | bne | | $[PC] + = ([GOP_1] \neq [GOP_2])? [imm] : 4$ |
| 000110 | blez | BranchZ | $[PC] + = ([GOP_1] \leq 0)? [imm] : 4$ |
| 000111 | bgtz | | $[PC] + = ([GOP_1] > 0)? [imm] : 4$ |
| 000010 | j | Jump | $[PC] + = [imm]$ |
| 000010 | jal | JumpAL | $[PC] + = [imm]$ $[G31] = [PC] + 4$ |
| 111111 | rfe | RFE | $[PC] = [S2]$ $[S0] = [S1]$ |

| $IR[31 : 26] = 000000$ | | | |
|------------------------|--------|---------|---|
| IR[5:0] | Instr. | Group | Effect |
| 001000 | jr | JumpR | $[PC] + = [GOP_1]$ |
| 001001 | jalr | JumpALR | $[PC] + = [GOP_1]$ $[GD] = [PC] + 4$ |

| $IR[31 : 26] = 000001$ | | | |
|------------------------|--------|-----------|--|
| IR[20:16] | Instr. | Group | Effect |
| 00000 | bltz | BranchZ | $[PC] + = ([GOP_1] < 0)? [imm] : 4$ |
| 00001 | bgez | | $[PC] + = ([GOP_1] \geq 0)? [imm] : 4$ |
| 10000 | bltzal | BranchZAL | $[PC] + = ([GOP_1] < 0)? [imm] : 4$ $[G31] = [PC] + 4$ |
| 10001 | bgezal | | $[PC] + = ([GOP_1] \geq 0)? [imm] : 4$ $[G31] = [PC] + 4$ |

| $IR[31 : 26] = 010001 \wedge IR[25 : 21] = 01000$ | | | |
|---|--------|-------|----------------------------------|
| IR[20:16] | Instr. | Group | Effect |
| 00000 | BC1F | BC1 | $[PC] + = ([S7] = 0)? [imm] : 4$ |
| 00001 | BC1T | | $[PC] + = ([S7] = 1)? [imm] : 4$ |

Table A.7: Control flow change instructions

| Group | [31:26] | [25:21] | [20:16] | [15:11] | [10:6] | [5:0] |
|-----------|------------------|----------------|----------|---------|--------|------------------|
| Shift | 000000 | OP_1 | OP_2 | D | | 00011* 0001*0 |
| ShiftI | 000000 | | OP_2 | D | imm | 00001* 0000*0 |
| Alu | 000000 | OP_1 | OP_2 | D | | 100*** 10101* |
| AluI | 001*** | OP_1 | D | imm | | |
| MoveF2I | 010001 | 00000 | D | OP_2 | | |
| MoveS2I | 010001 | 00010 | D | OP_2 | | |
| MoveI2F | 010001 | 00100 | OP_2 | D | | |
| MoveI2S | 010001 | 00110 | OP_2 | D | | |
| trap | 111110 | imm | | | | |
| Mult | 000000 | OP_1 | OP_2 | | | 01100* |
| Div | 000000 | OP_1 | OP_2 | | | 01101* |
| Load | 100*0* 1000*1 | OP_1 | D | imm | | |
| LoadLR | 100*10 | OP_1 | OP_2/D | imm | | |
| LoadFP | 110001 | OP_1 | D | imm | | |
| Store | 1010** 101110 | OP_1 | OP_2 | imm | | |
| StoreFP | 111001 | OP_1 | OP_2 | imm | | |
| Branch | 00010* | OP_1 | OP_2 | imm | | |
| BranchZ | 00011* 000001 | OP_1 | 0000* | imm | | |
| BranchZAL | 000001 | OP_1 | 1000* | imm | | |
| Jump | 000010 | imm | | | | |
| JumpAL | 000011 | imm | | | | |
| JumpR | 000000 | OP_1 | | | | 001000 |
| JumpALR | 000000 | OP_1 | | D | | 001001 |
| BC1 | 010001 | 01000 | 0000* | imm | | |
| rfe | 111111 | | | | | |
| FAddSub | 010001 | 1000* | OP_1 | OP_2 | D | 00000* |
| FMulDiv | 010001 | 1000* | OP_1 | OP_2 | D | 00001* |
| FComp | 010001 | 1000* | OP_1 | OP_2 | | 11**** |
| FMisc | 010001 | 1000* | | OP_2 | D | 0001*1 00011* |
| FCvt.W | 010001 | 1000* | | OP_2 | D | 100100 |
| FCvt.S | 010001 | 10001 10100 | | OP_2 | D | 100000 |
| FCvt.D | 010001 | 1000* 10100 | | OP_2 | D | 100001 |

Table A.8: Instruction set architecture encoding

Appendix B

Emulation of a MIPS R3000

Apart from interrupt handling, the ISA of the $DLX_{\pi+}$ differs in the following points from the MIPS R3000 processor ISA [KH92]:

- The $DLX_{\pi+}$ does not support delayed branch. The semantic of branch instructions are altered accordingly.
- The $DLX_{\pi+}$ does not have a co-processor 0 (memory management unit). CP0 instructions cause an illegal instruction interrupt.
- All special registers (including floating point special registers and the registers HI and LO) are combined in the special register file. Instructions accessing special registers must be emulated by the instructions *moves2i* respectively *movei2s*.
- The floating point special register is divided into the special register *SR* (floating point mask), *RM* (rounding mode), *IEEEf* (floating point flags), and *FCC* (floating point condition code). Instructions accessing the floating point special register are assumed to access the special register *IEEEf*.

The MIPS R3000 instructions which cannot be mapped directly are emulated as summarized in table B.1.

| MIPS instruction | $DLX_{\pi+}$ instruction |
|--|------------------------------|
| syscall,break | trap |
| cfc1,mflo,mflo | moves2i ($OP_1 = 6, 8, 9$) |
| ctc1,mtlo,mthi | movei2s ($D = 6, 8, 9$) |
| mfc1 | movef2i |
| mtc1 | movei2f |
| bc0,cfc0,mfc0,mtc0,tlbr,tlbwi,tlbwr,tlbp | not implemented |

Table B.1: Translation of MIPS R3000 instructions

Appendix C

Additional Circuits

In this chapter all additional circuits needed for the design of the $DLX_{\pi+}$ are described.

C.1 Basic Circuits

C.1.1 Design

Figure C.1 depicts the design of a half-unary find-last-one circuit. Note that the circuit computes additionally to the find-last-one output flo an output $zero$ which indicates that all input signals are zero. For $n = 1$ the design of the circuit is simple. For $n > 1$ the circuit is build from two half-unary find-last-one sub-circuits with only half of the input bits. One circuit uses the upper half of the input bits, the other the lower half. The zero output is active if both zero outputs of the sub-circuit are active. If the lower parts of the inputs contains a one (i.e., the zero output of the lower part is not active, the output flo of the upper part must be forced to zero.

C.1.2 Cost and Delay

This section lists the formulas for the cost and delay of the basic circuits used in this thesis.

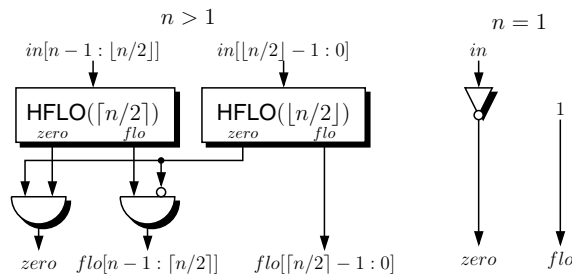


Figure C.1: Design of a half-unary find-last-one circuit

Find-Last-One / Find-First-One

$$\begin{aligned}
D(\text{FLO}(n)) &\leq \lceil \log n \rceil \cdot D_{AND}, \\
C(\text{FLO}(n)) &\leq \begin{cases} 0 & \text{if } n \leq 1 \\ C(\text{FLO}(\lceil n/2 \rceil)) + C(\text{FLO}(\lfloor n/2 \rfloor)) \\ \quad + (\lceil n/2 \rceil + 1) \cdot C_{AND} & \text{if } n > 1 \end{cases}, \\
D(\text{FFO}(n)) &\leq D(\text{FLO}(n)), \\
C(\text{FFO}(n)) &\leq C(\text{FLO}(n)), \\
D(\text{HFLO}(n)) &\leq D(\text{FLO}(n)), \\
C(\text{HFLO}(n)) &\leq C(\text{FLO}(n)),
\end{aligned}$$

Decoder / Encoder

$$\begin{aligned}
D(\text{Dec}(n)) &\leq \lceil \log n \rceil \cdot D_{AND}, \\
C(\text{Dec}(n)) &\leq \begin{cases} 0 & \text{if } n \leq 1 \\ C(\text{Dec}(\lceil n/2 \rceil)) + C(\text{Dec}(\lfloor n/2 \rfloor)) + 2^n \cdot C_{AND} & \text{if } n > 1 \end{cases}, \\
D(\text{HDec}(n)) &\leq n \cdot \max\{D_{AND}, D_{OR}\}, \\
C(\text{HDec}(n)) &\leq \begin{cases} 0 & \text{if } n \leq 1 \\ C(\text{HDec}(n-1)) + 2^{n-1} \cdot (C_{AND} + C_{OR}) & \text{if } n > 1 \end{cases}, \\
D(\text{Enc}(n)) &\leq \begin{cases} 0 & \text{if } n \leq 1 \\ D(\text{Enc}(\lceil n/2 \rceil)) + D_{MUX} & \text{if } n > 1 \end{cases}, \\
C(\text{Enc}(n)) &\leq \begin{cases} 0 & \text{if } n \leq 1 \\ C(\text{Enc}(\lceil n/2 \rceil)) + C(\text{Enc}(\lfloor n/2 \rfloor)) \\ \quad + C_{AND} + (\lfloor n/2 \rfloor) \cdot C_{MUX} & \text{if } n > 1 \end{cases}.
\end{aligned}$$

Parallel Prefix

$$\begin{aligned}
D(\text{PP-FUNC}(D(\text{FUNC}), n)) &\leq \lceil \log n \rceil \cdot D(\text{FUNC}), \\
C(\text{PP-FUNC}(C(\text{FUNC}), n)) &\leq \begin{cases} 0 & \text{if } n \leq 1 \\ C(\text{PP-FUNC}(\lceil n/2 \rceil)) \\ \quad + C(\text{PP-FUNC}(\lfloor n/2 \rfloor)) \\ \quad + (n-1) \cdot C(\text{FUNC}) & \text{if } n > 1 \end{cases}, \\
D(\text{PP-AND}(n)) &\leq D(\text{PP-FUNC}(D_{AND}, n)), \\
C(\text{PP-AND}(n)) &\leq C(\text{PP-FUNC}(C_{AND}, n)), \\
D(\text{PP-OR}(n)) &\leq D(\text{PP-FUNC}(D_{OR}, n)), \\
C(\text{PP-OR}(n)) &\leq C(\text{PP-FUNC}(C_{OR}, n)),
\end{aligned}$$

Incrementer / Adder

$$D(\text{Inc}(n)) \begin{cases} 0 & \text{if } n \leq 1 \\ \leq D(\text{PP-AND}(n-1)) + D_{XOR} & \text{if } n > 1 \end{cases},$$

$$C(\text{Inc}(n)) \leq C(\text{PP-AND}(n-1)) + (n-1) \cdot C_{XOR},$$

$$D(\text{Add}(n)) \leq D_{XOR} + D(\text{PP-FUNC}(D_{MUX}, n-1)) + \max\{D_{MUX}, D_{XOR}\},$$

$$C(\text{Add}(n)) \leq C(\text{PP-FUNC}(C_{MUX}, n)) + 2 \cdot n \cdot C_{XOR}.$$

Trees

$$D(\text{FUNC-Tree}(D(\text{FUNC}), n)) \leq \lceil \log n \rceil \cdot D(\text{FUNC}),$$

$$C(\text{FUNC-Tree}(C(\text{FUNC}), n)) \leq (n-1) \cdot C(\text{FUNC}),$$

$$D(\text{AND-Tree}(n)) \leq D(\text{FUNC-Tree}(D_{AND}, n)),$$

$$C(\text{AND-Tree}(n)) \leq C(\text{FUNC-Tree}(D_{AND}, n)),$$

$$D(\text{OR-Tree}(n)) \leq D(\text{FUNC-Tree}(D_{OR}, n)),$$

$$C(\text{OR-Tree}(n)) \leq C(\text{FUNC-Tree}(D_{OR}, n)).$$

Checks

$$D(\text{EQ}(n)) \leq D_{XOR} + D(\text{AND-Tree}(n)),$$

$$C(\text{EQ}(n)) \leq n \cdot C_{XOR} + C(\text{AND-Tree}(n)),$$

$$D(\text{Zero}(n)) \leq D(\text{OR-Tree}(n)),$$

$$C(\text{Zero}(n)) \leq C(\text{OR-Tree}(n)).$$

Selection Circuit

$$D(\text{Sel}(n)) \leq D_{AND} + D(\text{OR-Tree}(n)),$$

$$C(\text{Sel}(n)) \leq n \cdot C_{AND} + C(\text{OR-Tree}(n)),$$

C.2 Instruction Decode

The instructions are decoded by the two circuits **DestCmp** and **Decode**. The circuit **DestCmp** computes the bus $D.\star$ which contains all signals regarding the destination address. Since these signals are needed to update the producer tables, they are considered timing critical. The circuit **DestCmp** is therefore delay optimized. The remaining control signals are computed by the circuit **Decode**.

C.2.1 Decode

This section only describes the control signals which are needed by the decode phase. The computation of the control signals used by the functional units is described in the together with the functional units.

The main purpose of the circuit **Decode** is to compute the following busses: $FUtmp.\{IAlu, IMul, Mem, BCU, FAdd, FMul, FMisc\}$, $OP_1.\{gpr, fpr, spr\}$, and $OP_2.\{gpr, fpr, spr\}$. These busses define which FU should be used by an instruction and from which register file the operands have to be taken. At most one of a signal of a bus may be active. If none of the signals $OP_1.\star$ or $OP_2.\star$ is active, an immediate constant is used as operand.

The computation of the control signals is based on the division of the instructions into instruction groups. The instruction group of a instruction is defined in table A.8 along with the encoding for the group. Table C.1 shows for each instruction group which control signal must be active.

The circuit **Decode** computes for each instruction to which instruction group it belongs. The control signals are computed as or of the instruction groups in which the signal is active. The last column of the table C.1 defines for which group additional control signals needed in the decode phase have to be valid.

If an instruction is a floating point instruction (i.e., $IR[31 : 25] = 0100011$), but belongs to no group, the signal $FUunimp$ must be activated. If the instruction is not a floating point instruction and belongs to no group, the signal illegal instruction interrupt *ill* must be activated.

If the branch prediction assumes that an instruction is an control flow instruction (indicated by $IFQ.pb$), the instruction has to be sent to the branch checking unit in any case. Hence, the signal $FU.BCU$ has to be active and the remaining signals of the bus $FU.\star$ must be 0. If $IFQ.pb$ is not active, the value of $FUtmp.\star$ can be used to compute $FU.\star$.

If the instruction is not an control flow instruction, the signal fpb is activated to indicate an falsely predicted branch. If an instruction is a branch instruction but $IFQ.pb$ is not active, the signal npb (not-predicted branch) is activated. The signals fpb and npb are needed by the BCU. It follows:

$$\begin{aligned} FU.BCU &= FUtmp.BCU \vee IFQ.pb \\ FU.\star &= FUtmp.\star \wedge \overline{IFQ.pb} \\ fpb &= IFQ.pb \wedge \overline{FUtmp.BCU} \\ npb &= \overline{IFQ.pb} \wedge FUtmp.BCU \end{aligned}$$

The operands may only be double precision values, if the instruction is an floating point instruction. For floating point instructions double precision operands are indicated by the bit $IR[21]$. If the first operand is an immediate constant, the constant is always determined by the bits $IR[10 : 6]$. The immediate constant for operand 2 is the address of the instruction if an instruction memory interrupt occurred, otherwise the

| Group | $FUtmp.$ | $OP_1.$ | $OP_2.$ | |
|-----------|----------|---------|---------|-----------|
| Alu | IAlu | gpr | gpr | |
| AluI | | gpr | | |
| Shift | | gpr | gpr | |
| ShiftI | | | gpr | |
| MoveS2F | | | fpr | |
| MoveF2S | | | gpr | |
| MoveS2I | | | spr | |
| MoveI2S | | | gpr | |
| Trap | | | | $trap$ |
| Mult | IMul | gpr | gpr | |
| Div | | gpr | gpr | |
| Load | Mem | gpr | | |
| LoadLR | | gpr | gpr | |
| LoadFP | | gpr | | |
| Store | | gpr | gpr | |
| StoreFP | | gpr | fpr | $storeFP$ |
| Branch | BCU | gpr | gpr | |
| BranchZ | | gpr | | |
| BranchZAL | | gpr | | |
| Jump | | | | |
| JumpAL | | | | |
| JumpR | | gpr | | |
| JumpALR | | gpr | | |
| BC1 | | spr | | |
| RFE | | spr | spr | rfe |
| FAdd | FAdd | fpr | fpr | |
| FMul | FMul | fpr | fpr | |
| FComp | FMisc | fpr | fpr | |
| FMisc | | | fpr | |
| FCvt.W | | | fpr | |
| FCvt.S | | | fpr | |
| FCvt.D | | | fpr | |

Table C.1: Active control signals

sign extended bits $IR[15 : 0]$.

$$\begin{aligned}
 OP_{1,2}.dbl &= IR[21], \\
 OP_1.imm &= IR[10 : 6], \\
 OP_2.imm &= \begin{cases} fPC & \text{if } Ip f \vee Imal \\ IR[15]^{16} IR[15 : 0] & \text{if } \overline{Ip f} \wedge \overline{Imal} \end{cases}
 \end{aligned}$$

The only instructions that explicitly read or write the special register $IEEEf$ are *moves2i* respectively *movei2s*. These instruction access the register $IEEEf$ if the

| Variable | Meaning | Value |
|-----------------|---|-------|
| λ_{max} | length of longest monomial in table A.8 | 17 |
| λ_{sum} | accumulated length of all monomials in table A.8 | 395 |
| ν_{max} | maximum number of monomials per group in table A.8 | 2 |
| ν_{sum} | number of monomials in table A.8 | 41 |
| γ | number of groups in table C.1 | 32 |
| β_{max} | maximum frequency of a control signal in table C.1 | 15 |
| β_{sum} | accumulated frequency of all control signals in table C.1 | 70 |
| ω | number of output signals in table C.1 | 16 |

Table C.2: Variables of the Decode Computation

bits $IR[15 : 11]$ encode the number of this register (6):

$$\begin{aligned} readIEEEf &= MoveS2I \wedge IR[15 : 11] = 00110 \\ writeIEEEf &= MoveI2S \wedge IR[15 : 11] = 00110 \end{aligned}$$

Cost and delay of the circuit **Decode** can be computed using the variables from table C.2 derived from the tables A.8 and C.1:

$$\begin{aligned} C(\text{Decode}) &\leq (\lambda_{sum} - \nu_{sum}) \cdot C_{AND} + (\nu_{sum} - \gamma + \beta_{sum} - \omega) \cdot C_{OR} \\ &\quad + (12 + 8 + 7) \cdot C_{AND} + C_{OR} + 32 \cdot C_{MUX} \\ &\leq 381 \cdot C_{AND} + 64 \cdot C_{OR} + 32 \cdot C_{MUX} \\ D(\text{Decode}) &\leq \lceil \log \lambda_{max} \rceil \cdot D_{AND} + \lceil \log(\nu_{max} \cdot \beta_{max}) \rceil \cdot D_{OR} + D_{AND} \\ &\leq 6 \cdot D_{AND} + 5 \cdot D_{OR} \end{aligned}$$

C.2.2 Destination computation

The computation of the signals $D.\{gpr, fpr, spr\}$, $D.addr$, and $D.dbl$ can be derived from Table C.3. To avoid unnecessarily many stages of forwarding in the decode sub-phase $D1$ the delay of the circuit **DestCmp** should be minimized.

The computation of the signals can be simplified using the fact, that the illegal instruction signal ill causes an abort interrupt. Thus the value of the write signals may be arbitrary for illegal instructions. For the implementation $D.\mathfrak{R}.impl.write$ of a write signal $D.\mathfrak{R}.write$ of a register file \mathfrak{R} it suffices if the following equation holds:

$$D.\mathfrak{R}.write \leq D.\mathfrak{R}.impl.write \leq DR.\mathfrak{R}.write \vee ill$$

The other signals for the register file \mathfrak{R} $D.\mathfrak{R}.\star$ (excluding the write signal) need only to have correct values if the write signal is active. This simplifies the condition for the implementation of these signals $D.\mathfrak{R}.impl.\star$ as follows:

$$D.\mathfrak{R}.impl.\star = \begin{cases} D.\mathfrak{R}.\star & \text{if } D.\mathfrak{R}.write = 1 \\ \star & \text{else} \end{cases}$$

For the sake of simplicity the implementation of the above signals and their definition will be identified. Figures C.2 to C.2 show the parts of the circuit **DestCmp** for

| Group | $D.$ | $D.addr$ | $D.dbl$ |
|-----------|-------|---------------|----------|
| MoveI2F | fpr | $IR[20 : 16]$ | 0 |
| LoadFP | | $IR[10 : 6]$ | |
| FCvt.W | | | |
| FCvt.S | | | |
| FAddSub | | | $IR[21]$ |
| FMulDiv | | | |
| FMisc | | 1 | |
| FCvt.D | | | |
| AluI | gpr | $IR[20 : 16]$ | 0 |
| MoveF2I | | | |
| MoveS2I | | | |
| Load | | | |
| LoadLR | | $IR[15 : 11]$ | |
| Alu | | | |
| Shift | | | |
| ShiftI | | | |
| JumpALR | | | |
| BranchZAL | | 11111 | |
| JumpAL | | | |
| Mult | spr | 01000 | 1 |
| Div | | $IR[15 : 11]$ | |
| MoveI2S | | | |
| rfe | | | 0 |
| FComp | | | |

Table C.3: Destination registers

GPR, FPR, and SPR. Cost and delay of the circuit are:

$$D(\text{DestCmp}) \leq \max\{2 \cdot D_{OR}, 2 \cdot D_{AND}, D_{MUX}\} + 2 \cdot D_{OR}$$

$$C(\text{DestCmp}) \leq 58 \cdot C_{AND} + 16 \cdot C_{OR} + 27 \cdot C_{MUX}$$

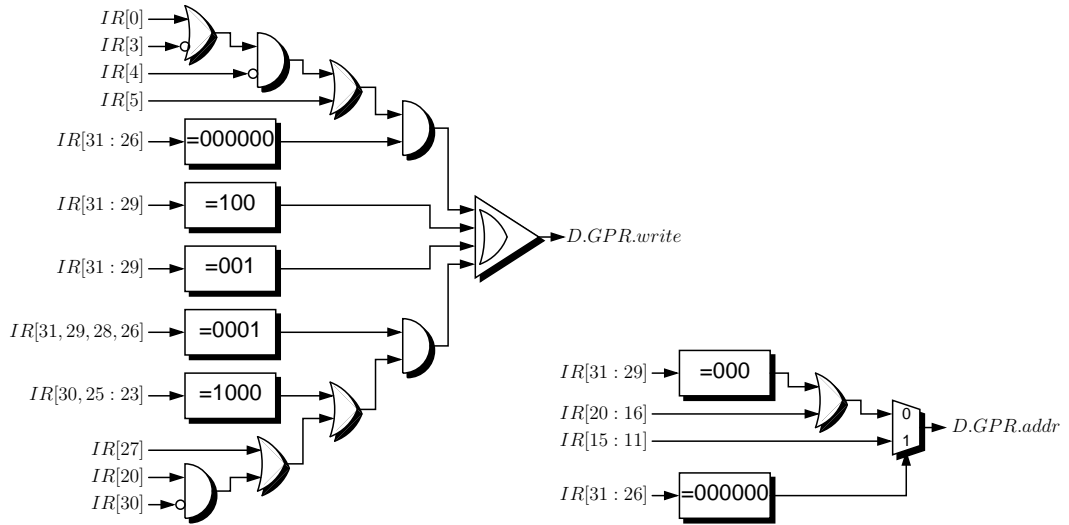


Figure C.2: Destination computation for GPR

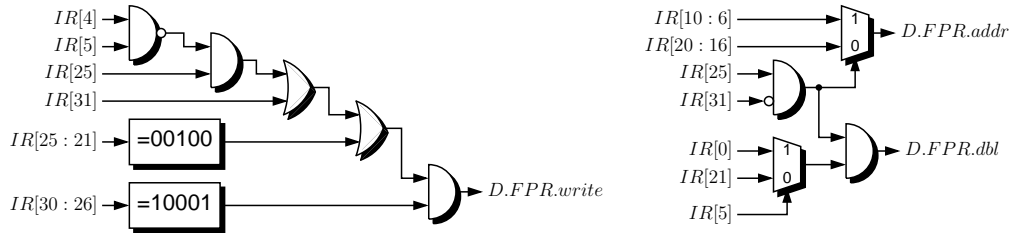


Figure C.3: Destination computation for FPR

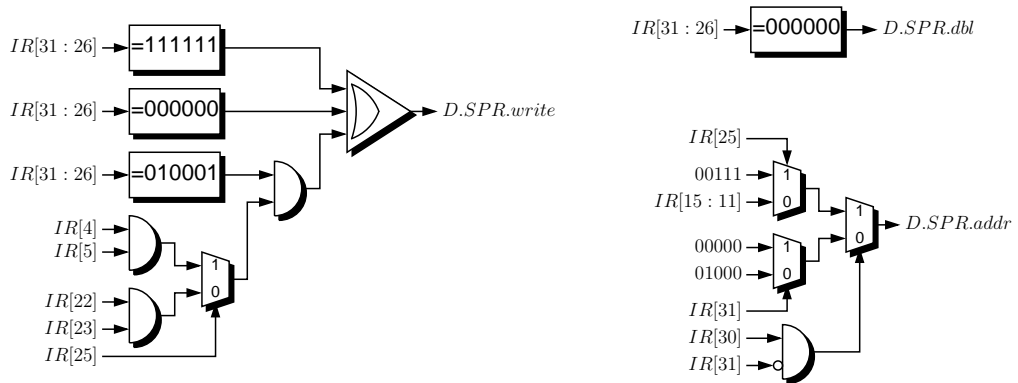


Figure C.4: Destination computation for SPR

Appendix D

Functional Units

The pipelining of the functional units is straightforward. The functional units used by the $DLX_{\pi+}$ are only listed for completeness and to compute the costs and number of cycles for the computations. The design of the integer multiplicative unit and the floating point units is not detailed here as it lies beyond the scope of this thesis.

D.1 Integer ALU

Figure D.1 depicts an overview of the integer ALU. It consists of an arithmetic unit AU, a shift unit SU, and a logic unit LU.

The AU is detailed in figure D.2. It performs additions, subtractions and test operations. The AU consists of an 32 bit adder and some glue logic to compute subtractions and the correct negative and overflow signals. Cost and delay of the arithmetic unit are:

$$\begin{aligned} D(\text{AU}) &\leq D(\text{Add}(32)) + 2 \cdot D_{XOR}, \\ C(\text{AU}) &\leq C(\text{Add}(32)) + 36 \cdot C_{XOR} + 2 \cdot C_{AND}. \end{aligned}$$

The SU (see figure D.3) uses a cyclic shifter to compute right and left shifts. The shift amount for right shifts is computed by an incrementer. A mask replaces the most significant bits by zero of the sign bit for arithmetic shifts. Cost and delay of the shift

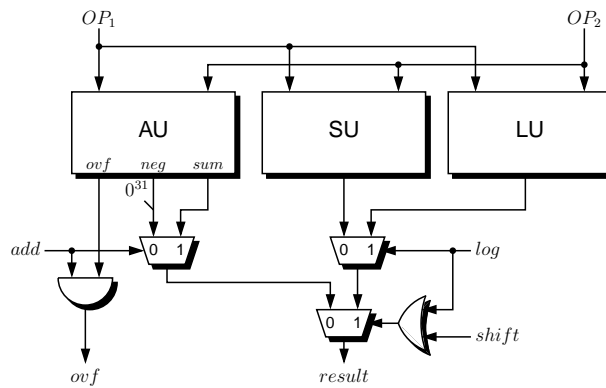


Figure D.1: Integer ALU

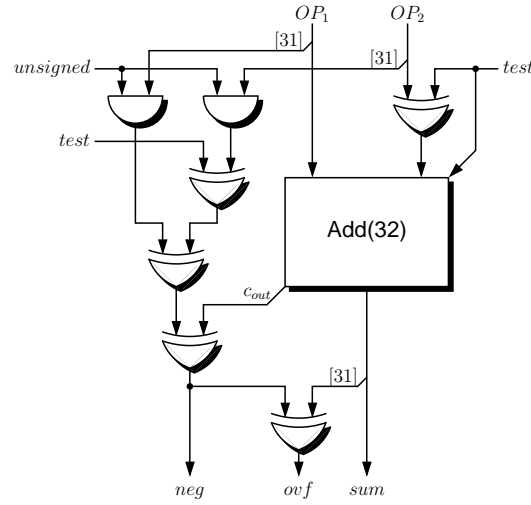


Figure D.2: Arithmetic unit

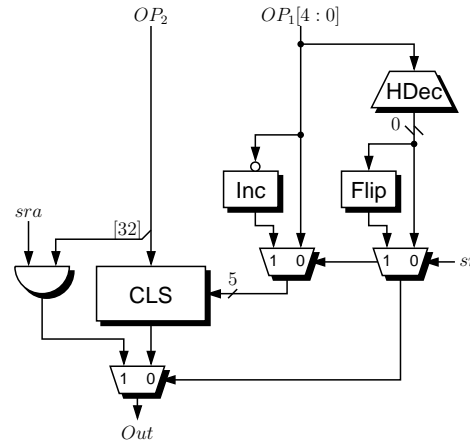


Figure D.3: Shift unit

unit are computed as:

$$\begin{aligned} D(\text{SU}) &\leq 7 \cdot D_{MUX}, \\ C(\text{SU}) &\leq C(\text{Inc}(5)) + 229 \cdot D_{MUX} + C_{AND}. \end{aligned}$$

The LU (see figure D.4) computes logic operations on the operands depending on the opcode. It also copies the second operand to the output for trap instructions. Cost and delay of the logic unit are:

$$D(\mathbf{LU}) \leq D_{XOR} + D(\mathbf{Sel}(5)),$$

$$C(\mathbf{LU}) \leq 64 \cdot C_{AND} + 32 \cdot C_{OR} + 32 \cdot C_{XOR} + 32 \cdot C_{NOR} + 32 \cdot C(\mathbf{Sel}(5)).$$

Let e_{RS_2} be the number of entries of the integer ALU reservation station and let $D(\text{FU}(e_{RS_2}))^+$ be the additional delay to the integer ALU from the reservation station. Let n be the number of functional units and let t_L be computed as in section 4.4. Then the delay of the integer ALU and the delay of the stall input from the completion phase

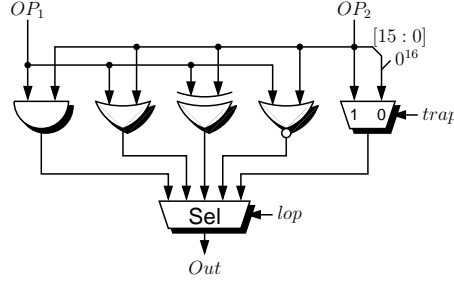


Figure D.4: Logic unit

are:

$$D(IALU) \leq D(FU(e_{RS_2}))^+ + \max\{D(AU), D(SU), D(LU)\} + 2 \cdot D_{MUX},$$

$$D(IALU.stallIn) \leq \begin{cases} D_{AND} + D(FLO(\min\{t_L, n\})) + D_{MUX} & \text{if } t_L > 2 \\ D_{AND} + D_{OR} & \text{if } t_L = 2 \end{cases}.$$

Let the variable b_{IALU} be one if a buffer circuit needs to be added to the integer ALU. This is the case if the following equation does not hold (similar to the BCU, see section 6.5.2):

$$\delta \geq \max\{D(AND-Tree(\lceil D(ALU)/\delta \rceil + 1)), D(IALU.stallIn)\} + D_{AND}$$

$$+ \begin{cases} 2 \cdot D_{AND} & \text{if } e_{RS_1} = 1 \\ 2 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS_1} > 1 \end{cases}$$

Hence, the number of cycles c_{IALU} and the cost of the integer ALU can be approximated by:

$$c_{IALU} = \lceil (D(ALU) + b_{IALU} \cdot D_{MUX})/\delta \rceil,$$

$$C(IALU) \leq C(AU) + C(SU) + C(LU) + 96 \cdot D_{MUX} + C_{AND} + C_{OR}$$

$$+ (c_{ALU} - 1) \cdot \lceil (65 + 8 + 34 + l_{ROB})/2 \rceil \cdot C_{REG}.$$

D.2 Integer Multiplicative Unit

Figure D.5 depicts the integer multiplicative unit. The first part of the unit distributes the instructions to either the multiplication or the division part of the unit. It also computes the booth recoding for the multiplier. The second part of the integer multiplicative unit consists of a Wallace tree for the multiplication and a SRT based divide circuit (see, e.g., [HOH97]). In the last part the carry-save respectively carry-borrow result of divisions or multiplications is compressed using a 64 bit adder. The design of the divider circuit is not detailed here. It is assumed that it has the same delay as the Wallace tree and computes 4 result digits. Thus, it has to be used 8 times to get the 32 bit quotient and remainder.

The delay of the distribution part is given by the cost and the delay of the booth recoding (see [MP00] for the formulas) and the additional delay introduced by the

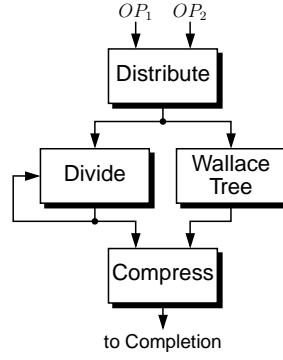


Figure D.5: Integer multiplicative unit

integer multiplicative reservation station. Let e_{RS_3} be the number of entries of the reservation station and $D(\text{FU}(e_{RS_3}))$ be the additional cost. Then:

$$C(\text{Distribute}) \leq 16 \cdot (2 \cdot C_{XOR} + C_{NOR}) + 33 \cdot (3 \cdot C_{NAND} + C_{XOR}),$$

$$D(\text{Distribute}) \leq D(\text{FU}(e_{RS_3})) + D_{XOR} + D_{NOR} + 2 \cdot D_{AND} + D_{XOR}.$$

The delay and cost of a Wallace Tree with 32 bits can also be derived from the formulas in [MP00]:

$$C(\text{WallaceTree}) \leq (35 \cdot 14 + 2 \cdot 2 \cdot 16) \cdot C(\text{Add}(1)),$$

$$D(\text{WallaceTree}) \leq 2 \cdot 3 \cdot D(\text{Add}(1)).$$

The cost of the divider are approximated by the cost of 4 32 bit carry-save adders and a 64 bit shifter.

$$C(\text{Divide}) \leq 4 \cdot 32 \cdot C(\text{Add}(1)) + 64 \cdot (C_{MUX} + C_{REG}).$$

The circuit **Compress** in the last part of the unit also selects between the outputs of the wallace tree and the divider circuit. Therefore, cost and delay of this circuit are approximated by:

$$D(\text{Compress}) \leq D_{MUX} + D(\text{Add}(64)),$$

$$C(\text{Compress}) \leq 64 \cdot C_{MUX} + C(\text{Add}(64)).$$

Buffer circuits are added to the first stages of the circuit **Compress** and **Distribute** if necessary. Let the variable b_{IMul3} be one if a buffer circuit is inserted into the circuit **Compress**, and b_{IMul1} be one if a buffer circuit is inserted into the circuit **Distribute**. Let $IMul.stallIn$ be the stall input of the integer multiplicative unit from the completion phase. Then the delay of the stall output to the reservation station (design not detailed here) is assumed to be:

$$D(IMul.stallOut) \leq \max\{\max\{D(\text{AND-Tree}(\lceil D(\text{Compress})/\delta \rceil)),$$

$$D(IMul.stallIn)\}, D(\text{AND-Tree}(\lceil D(\text{WallaceTree})/\delta \rceil))\}$$

$$+ D_{AND} + D_{OR} + D_{MUX} + D_{AND}.$$

If the equation

$$\delta \geq D(IMul.stallOut) + \begin{cases} 2 \cdot D_{AND} & \text{if } e_{RS_1} = 1 \\ 2 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS_1} > 1 \end{cases}$$

does not hold b_{IMul3} is set to one. This reduces the requirement to:

$$\delta \geq D(AND\text{-}Tree(\lceil D(WallaceTree)/\delta \rceil) + 2) + D_{OR} + \begin{cases} 2 \cdot D_{AND} & \text{if } e_{RS_1} = 1 \\ 2 \cdot D_{AND} + D_{MUX} & \text{if } e_{RS_1} > 1 \end{cases}$$

If this equation does not hold b_{IMul1} is also set to one. Then all requirements are assumed to hold for $\delta \geq 5$. Let c_{IMul1} , c_{IMul2} , c_{IMul3} be the number of cycles needed for the circuits **Distribute**, **WallaceTree**, and **Compress**. It holds:

$$\begin{aligned} c_{IMul1} &= \lceil (D(\text{Distribute}) + b_{IMul1} \cdot D_{MUX})/\delta \rceil, \\ c_{IMul2} &= \lceil D(\text{Multiply})/\delta \rceil, \\ c_{IMul3} &= \lceil (D(\text{Compress}) + b_{IMul3} \cdot D_{MUX})/\delta \rceil, \\ C(IMul) &\leq C(\text{Distribute}) + C(\text{Divide}) + C(\text{WallaceTree}) + C(\text{Compress}) \\ &\quad + (c_{IMul1} + c_{IMul2} + c_{IMul3}) \cdot \lceil (65 + 8 + 34)/2 \rceil \cdot C_{REG}. \end{aligned}$$

D.3 Floating Point Units

The delay of the additive and multiplicative floating point unit are from [Sei03]. The delay of the miscellaneous floating point unit is computed by synthesis using Synergy [Cad97] from the Verilog source [Lei02] of the corresponding unit from [Jac02]. The additive and the miscellaneous floating point units have straight pipelines. Let e_{RS_4} and e_{RS_6} be the number of entries of the reservation stations for the additive and miscellaneous floating point units and let $D(FU(e_{RS_4}))$ and $D(FU(e_{RS_6}))$ be the additional delay introduced by the reservation stations. The delay of the additive and miscellaneous floating point units is approximated by:

$$\begin{aligned} D(FPAdd) &\leq D(FU(e_{RS_4}))^+ + 114, \\ D(FPMisc) &\leq D(FU(e_{RS_6}))^+ + 30. \end{aligned}$$

A buffer circuit is inserted into the first stage of the additive and miscellaneous floating point unit if the requirements for the stall input of the corresponding reservation stations do not hold. This is assumed to suffice for the miscellaneous floating point unit and a stage depth $\delta = 5$. However a second buffer circuit has to be inserted into the additive floating point unit if the following equation does not hold:

$$\delta \geq \max\{D(FPAdd.stallIn), D(AND\text{-}Tree(\lceil D(FPAdd)/\delta \rceil + 1))\} + D_{AND}.$$

Let b_{FPAdd} and b_{FPMisc} be the number of buffers inserted into the respective floating point units. Then the number of cycles c_{FPAdd} and c_{FPMisc} are approximated by:

$$\begin{aligned} c_{FPAdd} &= \lceil (D(FPAdd) + b_{FPAdd} \cdot D_{MUX})/\delta \rceil, \\ c_{FPMisc} &= \lceil (D(FPMisc) + b_{FPMisc} \cdot D_{MUX})/\delta \rceil. \end{aligned}$$

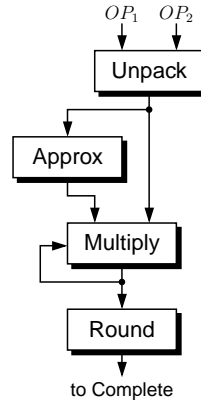


Figure D.6: Floating point multiplicative unit

An overview of the multiplicative floating point unit is depicted in figure D.6. The first part of the unit **Unpack** unpacks the operands. The approximation circuit for the reciprocal **Approx** is only needed by divisions. The third part of the unit is the actual multiplier circuit **Multiply**. While multiplication use this circuit only once, divisions need to use the circuit 5 times for single precision and 7 times for double precision. The last two multiplications of the divisions are independent and can be started in two succeeding cycles. The last part of the unit **Round** rounds the result. Let e_{RS_5} be the number of entries of the multiplicative floating point reservation station and let $D(\text{FU}(e_{RS_5}))$ be the additional delay for the functional unit from the reservation stations. Then the delays of the circuits are as follows [Sei03]:

$$\begin{aligned}
 D(\text{Unpack}) &\leq D(\text{FU}(e_{RS_5})) + 29, \\
 D(\text{Approx}) &\leq 38, \\
 D(\text{Multiply}) &\leq 67, \\
 D(\text{Round}) &\leq 34.
 \end{aligned}$$

The delay of the stall output (design not detailed here) if no buffer circuits are inserted is:

$$\begin{aligned}
 D(\text{stallOut}) &\leq \max\{D(\text{FPMul.stallIn}), D(\text{AND-Tree}(\lceil 34/\delta \rceil + \lceil 67/\delta \rceil + 1))\} \\
 &\quad + D_{AND} + D_{OR} + D_{AND} + D_{MUX} + D_{AND}.
 \end{aligned}$$

If this does not fulfill the requirements of the stall input of the reservation station, a buffer circuit is inserted into the first stage of the circuit **Multiply**. This reduces the delay of the stall output to:

$$\begin{aligned}
 D(\text{stallOut}) &\leq \max\{D_{AND} + D_{OR}, D(\text{AND-Tree}(\lceil 38/\delta \rceil))\} \\
 &\quad D_{AND} + D_{MUX} + D_{AND}.
 \end{aligned}$$

If this also does not fulfill the requirements another buffer circuit is inserted into the first stage of the circuit **Unpack**. Let c_{FPMul1} , c_{FPMul2} , c_{FPMul3} , and c_{FPMul4} be the number of cycles needed for the circuits **Unpack**, **Approx**, **Multiply**, and **Round**. Let the variables b_{Mul1} and b_{Mul3} indicated if buffer circuits are inserted into the

circuits **Unpack** and **Multiply**. Then:

$$\begin{aligned} c_{FPMul1} &= \lceil (D(\mathbf{Unpack}) + b_{FPMul1} \cdot D_{MUX}) / \delta \rceil, \\ c_{FPMul2} &= \lceil D(\mathbf{Approx}) / \delta \rceil, \\ c_{FPMul3} &= \lceil (D(\mathbf{Multiply}) + b_{FPMul3} \cdot D_{MUX}) / \delta \rceil, \\ c_{FPMul4} &= \lceil D(\mathbf{Round}) / \delta \rceil, \end{aligned}$$

The cost of the floating point circuits are approximated by the cost of the corresponding floating point units from [Lei99]. The number of inputs of the floating point units is $146 + l_{ROB}$, the number of outputs is $70 + l_{ROB}$. Then the costs of the units are approximated by:

$$\begin{aligned} D(\mathbf{FPAdd}) &\leq 23735 + (c_{FPAdd} - 1) \cdot (108 + l_{ROB}) \cdot C_{REG}, \\ D(\mathbf{FPMisc}) &\leq 15926 + (c_{FPMisc} - 1) \cdot (108 + l_{ROB}) \cdot C_{REG}, \\ D(\mathbf{FPMul}) &\leq 47557 + (c_{FPMul1} + c_{FPMul2} + c_{FPMul3} + c_{FPMul4} - 4) \\ &\quad \cdot (108 + l_{ROB}) \cdot C_{REG}. \end{aligned}$$

D.4 Memory Unit

This section describes the circuits shift for store **Sh4S** and shift for load **Sh4L** used by the memory unit.

D.4.1 Shift for Store

The shift for store circuit **Sh4S** mainly computes the effective address of the memory access and shifts the store data to the right position. The design of circuit **Sh4S** is straightforward except for the load / store word left / right instructions. Table D.1 shows the mapping of the source bytes to the destination bytes for these instructions in dependence of the bits 0 to 1 of the effective address. The bold numbers for the load instructions are the bytes of the destination register (which is also operand two), that may not be changed.

The data cache can replace a byte i only with the byte i of the word loaded from the memory. Thus, the operand two has to be shifted before the cache access in a way that the bytes which must not be replace are in the positions that are not written. The circuit **Sh4L** then shifts the bytes at the final position. Table D.2 shows the result of the data cache, if the bytes to be preserved are shifted at the positions that are not overwritten.

For default store instructions, the bytes to be written have to be shifted $\langle ea[1 : 0] \rangle$ bytes to the left. This also holds for store word right and load word right instructions. If cyclic shifters are used it suffices to pre-shift the data by one byte for load word left and store word left instruction.

Figure D.7 shows the circuit **Sh4S**. Apart from the effective address ea and the bus $data$ it computes the bus ub_* that indicates which bytes are used, and the data misaligned interrupt $Dmal$. The circuit uses multiple control signals which can be derived from the opcode and are not assumed to be timing critical. The signals lwl

is active, the instruction is a store / load word left / right instruction. Otherwise the signals hw and w indicate a half-word respectively *word* wide access.

For standard accesses the computation of the bus ub_* based on the signals $ea[1 : 0]$, hw , and w is straight-forward. For load / store word left / right instructions the bus can be computed with half-decoders. The interrupt $Dmal$ must be active if the bit $ea[0]$ is active and the instruction is a half-word or word-wide access or the bit $eq[1]$ is active for a word-wide access. The the data misaligned interrupt $Dmal$ is active the instruction must not be sent to the data cache but to the circuit **Sh4L**. Hence:

$$\begin{aligned} DCache.full &= full \wedge \overline{Dmal}, \\ Sh4L.full &= full \wedge Dmal. \end{aligned}$$

Note that only the lowest order bits of the effective address ea are used by the other circuits. The delay and cost of the circuit **Sh4S** are (including the computation of the control signals):

$$\begin{aligned} D(\text{Sh4S}) &\leq \max\{D(\text{Add}(32)), D(\text{Add}(2)) + D(\text{HDec}(2)) + 2 \cdot D_{MUX}\}, \\ C(\text{Sh4S}) &\leq C(\text{Add}(32)) + 2 \cdot C(\text{Dec}(2)) + C(\text{HDec}(2)) \\ &\quad + 104 \cdot C_{MUX} + 11 \cdot C_{OR} + 10 \cdot C_{AND}. \end{aligned}$$

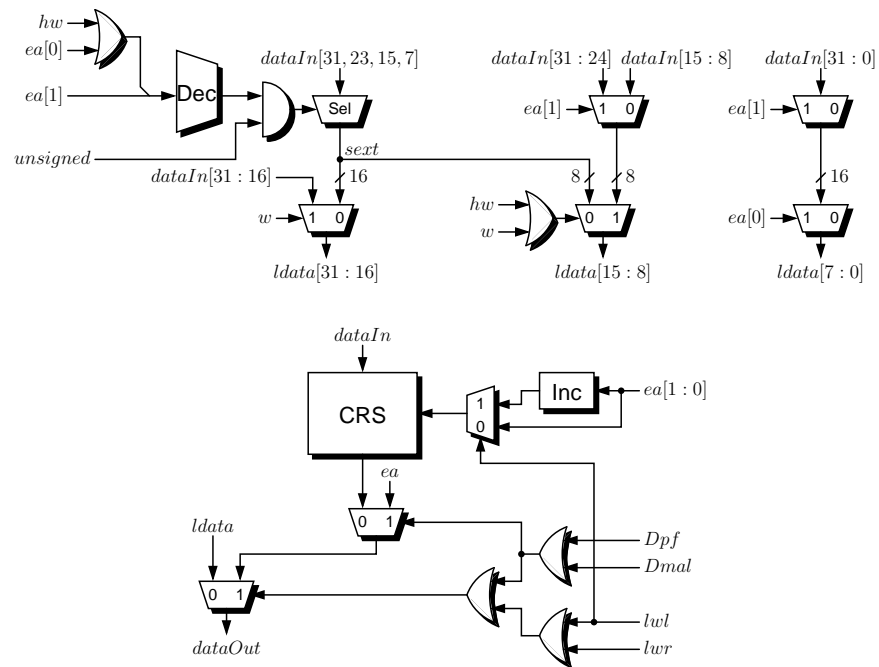
D.4.2 Shift for Load

The shift for load circuit **Sh4L** mainly receives the result from the data cache, shifts the bytes to be read to the right, and does a sign extension if necessary. The arbiter circuit to the memory unit which selects the source of the next data to be completed is described in the section 5.7 of the memory unit. Figure D.8 shows the reset of the shift for load circuit.

The upper part of the circuit computes the result (including sign extension) for standard read accesses. If the instruction is a load word left or load word right, the word must be shifted from the position in table D.2 to the position in table D.1. This can be done using a cyclic shifter and an incrementer. If an interrupt occurred, the memory unit must return the effective address as result.

Note that the control signals are not critical as they can be computed during shift for store. Then the delay and cost of the circuit **Sh4L** (including the computation of the control signals) are:

$$\begin{aligned} D(\text{Sh4L}) &\leq 4 \cdot D_{MUX}, \\ C(\text{Sh4L}) &\leq C(\text{Inc}(2)) + C(\text{Dec}(2)) + C(\text{Sel}(4)) \\ &\quad + 186 \cdot C_{MUX} + 3 \cdot D_{OR} + D_{AND}. \end{aligned}$$

**Figure D.8:** Shift for Load

Appendix E

Cost and Delay

This appendix lists the variables of the $DLX_{\pi+}$ in dependence of the parameters listed in table E.1. In order to reduce the number of dimensions only three parameters are assumed to be variable: the cycle time, the ROB size and the number of entries per reservation station. For all other parameters reasonable default values are chosen.

The other parameters are set as follows: The $DLX_{\pi+}$ has only one functional unit per type. The data and the instruction cache are both 4-way set associative, 16K large and have both 32 byte wide cache-lines. The update queue of the data cache has 4 entries, the read queue has 8 entries. The BTB is 4-way set associative and has 256 entries. The cache fetches 8 instructions per $IFCLK$ cycle.

Note that for a stage depth δ of 5 it is not possible to build an update queue or read queue with more than two entries. For $\delta = 6$ the read queue may have at most four entries. In these cases the number of entries of the queues are set to the maximum possible value.

The cycle time is between 10 and 100 gate delays (i.e., $5 \leq \delta \leq 95$). The ROB

| Variable | Description | Default |
|--------------------------|--|----------|
| $\tau := \delta + 5$ | cycle time of the $DLX_{\pi+}$ | 10 – 100 |
| $L_{ROB} := 2^{l_{ROB}}$ | # lines of the reorder buffer | 32 – 128 |
| n_i | # FU's of type i ($0 \leq i \leq 6$) | 1 |
| e_{RS_i} | # entries of the RSs of type i ($0 \leq i \leq 6$) | 2 – 8 |
| $L_{DC} := 2^{l_{DC}}$ | # lines of the data cache | 128 |
| $S_{DC} := 2^{s_{DC}}$ | # bytes of the cache-lines of the data cache | 32 |
| $K_{DC} := 2^{k_{DC}}$ | associativity of the data cache | 4 |
| e_{UQ} | # entries of the update queue | 4 |
| e_{RQ} | # entries of the read queue | 8 |
| $L_{IC} := 2^{l_{IC}}$ | # lines of the instr. cache | 128 |
| $S_{IC} := 2^{s_{IC}}$ | # bytes of the cache-lines of the instr. cache | 32 |
| $K_{IC} := 2^{k_{IC}}$ | associativity of the instruction cache | 4 |
| $L_{BTB} := 2^{l_{BTB}}$ | # lines of the branch target buffer | 64 |
| $K_{BTB} := 2^{k_{BTB}}$ | associativity of the branch target buffer | 4 |
| $FS := 2^{f_s}$ | # instructions fetched per cycle | 8 |

Table E.1: Parameters of the $DLX_{\pi+}$

has between 32 and 128 lines. The reservation stations have between 2 and 8 entries. For simplicity all reservation stations have the same number of entries. Note that for $\delta \in \{5, 6\}$ the all reservation stations except the memory and the branch checking reservation station may have at most two respectively four entries. For the memory and branch checking reservation station, the allowed number of entries is even less. Therefore, for these reservation stations also the results for only one entry is given. No results are given if the number of reservation station is larger than the maximum possible number.

Tables E.2 and E.3 list the variables of the $DLX_{\pi+}$ in dependence of the three free parameters. If the variables do not change if one of the parameters is increased, the line respectively column is omitted. Tables E.4 and E.5 list the overall cost of the $DLX_{\pi+}$ in dependence of the three free parameters.

Table E.2: Variables of the DLX _{π^+} (part 1)

| τ | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 25 | 27 | 29 | 33 | 34 | 35 | 37 | 41 | 49 | 50 | 67 | 77 | $e_{RS\star}$ | L_{ROB} |
|------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---------------|-----------|
| m | 5 | 5 | 5 | 4 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | * | * |
| c_{IF} | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | * | * |
| c_{D1} | 7 | 6 | 5 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * | * |
| c_{D2} | 6 | 5 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * | 32 |
| c_{D2} | 9 | 8 | 7 | 6 | 5 | 5 | 4 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | * | 64 |
| c_{D2} | 15 | 12 | 11 | 9 | 8 | 8 | 7 | 6 | 6 | 6 | 5 | 5 | 5 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 1 | * | 128 |
| c_I | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_I | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | * |
| c_I | na | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | * |
| c_I | na | na | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8 | * |
| c_T | 3 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * | * |
| c_{U2DM} | 3 | 3 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_{U2DM} | na | 4 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | * |
| c_{U2DM} | na | na | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | * |
| c_{U2DM} | na | na | na | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8 | * |
| c_{U2DI} | 4 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | * |
| c_{U2DI} | na | 4 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | * |
| c_{U2DI} | na | na | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8 | * |
| c_{U2DF} | 5 | 4 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | * |
| c_{U2DF} | na | 4 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | * |
| c_{U2DF} | na | na | 4 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8 | * |
| c_{U2DB} | 3 | 3 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_{U2DB} | na | na | na | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8 | * |
| c_{AT} | 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * | * |
| c_C | 3 | 3 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * | * |
| c_{C2R} | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * | * |
| c_{Ret2} | 7 | 6 | 5 | 5 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | * | * |

| τ | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 33 | 34 | 35 | 38 | 39 | 43 | 45 | 46 | 47 | 48 | 62 | 65 | 72 | e_{RS_*} |
|--------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|------------|
| c_{Sh4S} | 3 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| c_{Sh4S} | na | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| c_{Sh4S} | na | na | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| c_{Sh4S} | na | na | na | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 |
| c_{HC} | 12 | 10 | 9 | 8 | 7 | 6 | 6 | 5 | 5 | 5 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | * |
| c_{M2W} | 4 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_{M2R} | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_{M2Q} | 4 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_{Sh4L} | 3 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_{IALU} | 4 | 3 | 3 | 2 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | * |
| c_{IMul1} | 2 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| c_{IMul1} | na | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| c_{IMul1} | na | na | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 |
| c_{IMul2} | 6 | 5 | 4 | 4 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_{IMul3} | 4 | 3 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_{FPAdd} | 23 | 18 | 15 | 14 | 12 | 11 | 9 | 9 | 9 | 8 | 7 | 6 | 6 | 6 | 5 | 5 | 5 | 5 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | * |
| c_{FPMul1} | 6 | 5 | 4 | 4 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| c_{FPMul1} | na | 5 | 4 | 4 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 4 |
| c_{FPMul1} | na | na | 4 | 4 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 8 |
| c_{FPMul2} | 8 | 7 | 6 | 5 | 5 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_{FPMul3} | 14 | 12 | 10 | 9 | 8 | 7 | 7 | 6 | 6 | 5 | 5 | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | * |
| c_{FPMul4} | 7 | 5 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | * |
| c_{FPMisc} | 5 | 4 | 3 | 3 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | * |
| c_{BCU} | 5 | 3 | 3 | 2 | 2 | 1 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | * |

Table E.3: Variables of the DLX $_{\pi+}$ (part 2)

Table E.4: Cost of the DLX $_{\pi^+}$ (part 1)

| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | e_{RS^*} | L_{ROB} |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|------------|-----------|
| 2748823 | 2428247 | 2247879 | 2067335 | 1899064 | 1784354 | 1748697 | 1649984 | 1637588 | 1563675 | 1447715 | 2 | 32 |
| 2820714 | 2503079 | 2319473 | 2123671 | 1936419 | 1830683 | 1783910 | 1685063 | 1672439 | 1607267 | 1479595 | 2 | 64 |
| 2971889 | 2633813 | 2445418 | 2227592 | 2030252 | 1920692 | 1870585 | 1760683 | 1744375 | 1678605 | 1550465 | 2 | 128 |
| na | 2599696 | 2402933 | 2217131 | 2002000 | 1887290 | 1851633 | 1742998 | 1742062 | 1651431 | 1555667 | 4 | 32 |
| na | 2667841 | 2477693 | 2276523 | 2041597 | 1935861 | 1889088 | 1780209 | 1779257 | 1697045 | 1589943 | 4 | 64 |
| na | 2788361 | 2606804 | 2383500 | 2137672 | 2028112 | 1978005 | 1857961 | 1853537 | 1770405 | 1663209 | 4 | 128 |
| na | na | 2775631 | 2551181 | 2342078 | 2093720 | 2058503 | 1929120 | 1928184 | 1860937 | 1731273 | 8 | 32 |
| na | na | 2846436 | 2617345 | 2388535 | 2146783 | 2100450 | 1970595 | 1969643 | 1911247 | 1769593 | 8 | 64 |
| na | na | 2967405 | 2731094 | 2491470 | 2243526 | 2193859 | 2052611 | 2048187 | 1989303 | 1846903 | 8 | 128 |

| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | e_{RS^*} | L_{ROB} |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|------------|-----------|
| 1445719 | 1427559 | 1365538 | 1289197 | 1349252 | 1348348 | 1348348 | 1347444 | 1345292 | 1344820 | 1344820 | 2 | 32 |
| 1477599 | 1459407 | 1397386 | 1321037 | 1381084 | 1380172 | 1369087 | 1368175 | 1365999 | 1365519 | 1365519 | 2 | 64 |
| 1545013 | 1526789 | 1453850 | 1377493 | 1437532 | 1436612 | 1433156 | 1432236 | 1419113 | 1418625 | 1418625 | 2 | 128 |
| 1486615 | 1468919 | 1406434 | 1330093 | 1390148 | 1389244 | 1389244 | 1388340 | 1386972 | 1385716 | 1385716 | 4 | 32 |
| 1519703 | 1501983 | 1439490 | 1363141 | 1423188 | 1422276 | 1411191 | 1410279 | 1408895 | 1407623 | 1407623 | 4 | 64 |
| 1588325 | 1570581 | 1497162 | 1420805 | 1480844 | 1479924 | 1476468 | 1475548 | 1463225 | 1461937 | 1461937 | 4 | 128 |
| 1702613 | 1550805 | 1488784 | 1411979 | 1472034 | 1472034 | 1471130 | 1470226 | 1468858 | 1468386 | 1467602 | 8 | 32 |
| 1740493 | 1586285 | 1524264 | 1447443 | 1507490 | 1507490 | 1495493 | 1494581 | 1493197 | 1492717 | 1491925 | 8 | 64 |
| 1813907 | 1657299 | 1584360 | 1507523 | 1567562 | 1567562 | 1563186 | 1562266 | 1549943 | 1549455 | 1548655 | 8 | 128 |

Table E.5: Cost of the DLX _{π^+} (part 2)

| 32 | 33 | 34 | 35 | 37 | 38 | 39 | 41 | 43 | 45 | 46 | e_{RS^*} | L_{ROB} |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|------------|-----------|
| 1342495 | 1331144 | 1198225 | 1195777 | 1194133 | 1193229 | 1192325 | 1192325 | 1191421 | 1190517 | 1189613 | 2 | 32 |
| 1362985 | 1360961 | 1228042 | 1225562 | 1223918 | 1223006 | 1222094 | 1222094 | 1221182 | 1220270 | 1219358 | 2 | 64 |
| 1415884 | 1413844 | 1280925 | 1278405 | 1273305 | 1272385 | 1271465 | 1260547 | 1259627 | 1258707 | 1257787 | 2 | 128 |
| 1383391 | 1372040 | 1239121 | 1236673 | 1235029 | 1234125 | 1233221 | 1233221 | 1232317 | 1231413 | 1231413 | 4 | 32 |
| 1405089 | 1403065 | 1270146 | 1267666 | 1266022 | 1265110 | 1264198 | 1264198 | 1263286 | 1262374 | 1262374 | 4 | 64 |
| 1459196 | 1457156 | 1324237 | 1321717 | 1316617 | 1315697 | 1314777 | 1303859 | 1302939 | 1302019 | 1302019 | 4 | 128 |
| 1465277 | 1453926 | 1321007 | 1318559 | 1316915 | 1316011 | 1315107 | 1315107 | 1314203 | 1313299 | 1313299 | 8 | 32 |
| 1489391 | 1487367 | 1354448 | 1351968 | 1350324 | 1349412 | 1348500 | 1348500 | 1347588 | 1346676 | 1346676 | 8 | 64 |
| 1545914 | 1543874 | 1410955 | 1408435 | 1403335 | 1402415 | 1401495 | 1390577 | 1389657 | 1388737 | 1388737 | 8 | 128 |

| 47 | 48 | 49 | 50 | 62 | 65 | 67 | 69 | 72 | 77 | e_{RS^*} | L_{ROB} |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|------------|-----------|
| 1189613 | 1189613 | 1189613 | 1251634 | 1185290 | 1184386 | 1122365 | 1122365 | 1121461 | 1121461 | 2 | 32 |
| 1219358 | 1219358 | 1208272 | 1270293 | 1203949 | 1203037 | 1141016 | 1141016 | 1140104 | 1140104 | 2 | 64 |
| 1257787 | 1257787 | 1257787 | 1319808 | 1253464 | 1252544 | 1190523 | 1187067 | 1186147 | 1175224 | 2 | 128 |
| 1230509 | 1230509 | 1230509 | 1292530 | 1226186 | 1225282 | 1163261 | 1163261 | 1162357 | 1162357 | 4 | 32 |
| 1261462 | 1261462 | 1250376 | 1312397 | 1246053 | 1245141 | 1183120 | 1183120 | 1182208 | 1182208 | 4 | 64 |
| 1301099 | 1301099 | 1301099 | 1363120 | 1296776 | 1295856 | 1233835 | 1230379 | 1229459 | 1218536 | 4 | 128 |
| 1313299 | 1312395 | 1312395 | 1374416 | 1308072 | 1307168 | 1245147 | 1245147 | 1244243 | 1244243 | 8 | 32 |
| 1346676 | 1345764 | 1334678 | 1396699 | 1330355 | 1329443 | 1267422 | 1267422 | 1266510 | 1266510 | 8 | 64 |
| 1388737 | 1387817 | 1387817 | 1449838 | 1383494 | 1382574 | 1320553 | 1317097 | 1316177 | 1305254 | 8 | 128 |

Bibliography

- [BJK⁺03] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Instantiating uninterpreted functional units and memory system: functional verification of the VAMP. In *CHARME 2003*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.
- [BLM91] Asghar Bashteen, Ivy Lui, and Jill Mullan. A superpipeline approach to the MIPS architecture. In *Proceedings of the IEEE COMPCON*, pages 8–12, 1991.
- [Cad97] Cadence Design Systems Inc. *Synergy HDL Command Reference*, 1997.
- [Grü94] Thomas Grün. *Quantitative Analyse von I/O-Architekturen*. Dissertation (paul), Universität des Saarlandes, Computer Science Department, Saarbrücken, 1994.
- [GV95] Bernard Goossens and Duc Thang Vu. Further pipelining and multithreading to improve risc processor speed. a proposed architecture and simulation results. In *Proceedings of the 3rd International Conference of Parallel Computing Technologies*, pages 326–340, September 1995.
- [GV96] Bernard Goossens and Duc Thang Vu. Multithreading to improve cycle width and cpi in superpipelined superscalar processors. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, pages 36–42, June 1996.
- [HHM99] M. Horowitz, R. Ho, and K. Mai. The future of wires. In *Invited Workshop Paper for SRC Conference*, May 1999.
- [Hil95] Mark Hill. SPEC92 Traces for MIPS R3000 processors. University of Wisconsin, Madison, 1995.
<ftp://tracebase.nmsu.edu/pub/tracebase4/r3000/>.
- [Hil00] Mark A. Hillebrand. Design and evaluation of a superscalar risc processor. Diplomarbeit, Universität des Saarlandes, Computer Science Department, Saarbrücken, 2000.
- [HJF⁺02] M.S. Hrishikesh, Norman P. Jouppi, Keith I. Farkas, Doug Burger, Stephen W. Keckler, and Premkishore Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 14–24, May 2002.

- [HOH97] David L. Harris, Stuart F. Oberman, and Mark H. Horowitz. SRT division architectures and implementations. In *Proceeding of the 13th IEEE Symposium on Computer Arithmetic*, pages 18–25, 1997.
- [HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach; second edition*. Morgan Kaufmann, San Francisco, California, 1996.
- [HP02] A. Hartstein and Thomas R. Puzak. The optimum pipeline depth for a microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 7–13, May 2002.
- [HP03] A. Hartstein and Thomas R. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the 36th International Symposium on Microarchitecture*, 2003.
- [HSU⁺01] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, First Quarter 2001.
- [IEEE] Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.
- [Jac02] Christian Jacobi. *Formal Verification of a Fully IEEE Compliant Floating Point Unit*. Dissertation (paul), Universität des Saarlandes, Computer Science Department, Saarbrücken, 2002.
- [JCL94] Stephan Jourdan, Dominique Carrière, and Daniel Litaize. A high out-of-order issue symmetric superpipeline superscalar microprocessor. In *Proceedings of the 20th Euromicro conference*, pages 338–345, January 1994.
- [KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, New Jersey, 1992.
- [KMP99] Daniel Kröning, Silvia M. Müller, and Wolfgang Paul. A rigorous correctness proof of the Tomasulo scheduling algorithm with precise interrupts. In *Proceedings of the SCI'99/ISAS'99 International Conference*, 1999.
- [Kog81] Peter M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, 1981.
- [KP95] Jörg Keller and Wolfgang Paul. *Hardware Design — Formaler Entwurf Digitaler Schaltungen*. TEUBNER, 1995. (in German).
- [Kro81] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 81–87, May 1981.
- [Krö99] Daniel Kröning. Design and evaluation of a risc processor with a tomasulo scheduler. Diplomarbeit, Universität des Saarlandes, Computer Science Department, Saarbrücken, 1999.

- [Krö01] Daniel Kröning. *Formal Verification of Pipelined Microprocessors*. Dissertation (paul), Universität des Saarlandes, Computer Science Department, Saarbrücken, 2001.
- [KS86] Steven R. Kunkel and James E. Smith. Optimal pipelining in supercomputers. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 404–411, June 1986.
- [Lei99] Holger W. Leister. *Quantitative Analysis of Precise Interrupt Mechanisms for Out-Of-Order Execution Processors*. Dissertation (paul), Universität des Saarlandes, Computer Science Department, Saarbrücken, 1999.
- [Lei02] Dirk Leinenbach. Implementierung eines maschinell verifizierten prozessors. Diplomarbeit, Universität des Saarlandes, Computer Science Department, Saarbrücken, 2002. (in German).
- [LS84] Johnny K. F. Lee and Alan J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, January 1984.
- [McF93] Scott McFarling. Combining Branch Predictors. Technical Report TN-36, DEC Western Research Laboratory, June 1993.
- [MLD⁺99] Silvia M. Müller, Holger Leister, Peter Dell, Nikolaus Gerteis, and Daniel Kröning. The impact of hardware scheduling mechanisms on the performance and cost of processor designs. In *Proceedings of the 15th GI/ITG Conference 'Architektur von Rechensystemen' ARCS'99*, pages 65–73. VDE Verlag, 1999.
- [MP95] Silvia M. Müller and Wolfgang J. Paul. *The Complexity of simple Computer Architectures*. Springer, Berlin;Heidelberg;New York, 1995.
- [MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture, Complexity and Correctness*. Springer, Berlin;Heidelberg;New York, 2000.
- [PJS97] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, 1997.
- [SBG⁺02] Viji Srinivasan, David Brooks, Michael Gschwind, Pradip Bose, Victor Zyuban, Philip N. Strenski, and Philip G. Emma. Optimizing pipelines for power and performance. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 333–344, November 2002.
- [SC02] Eric Sprangle and Doug Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.
- [Sei99] Peter-Michael Seidel. *On the Design of IEEE Compliant Floating-Point Units and Their Quantitative Analysis*. Dissertation (paul), Universität des Saarlandes, Computer Science Department, Saarbrücken, 1999.

- [Sei03] Peter-Michael Seidel. Delay of IEEE Compliant Floating Point Units. Personal Communication, March 2003.
- [Sic92] James E. Siculo. A multiported nonblocking cache for a superscalar uniprocessor. Ms thesis, University of Illinois, Department of Computer Science, Urbana IL, 1992.
- [SP85] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 36–44, June 1985.
- [SPEC] Standard Performance Evaluation Corporation. *The SPEC92 benchmark suite*. <http://www.specbench.org>.
- [SSH99] Ivan Sutherland, Bob Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kauffman Publishers, 1999.
- [Tom67] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal Research and Development*, 11:25–33, January 1967.
- [Web88] C. F. Webb. Subroutine call/return stack. Technical Report 30(11), IBM Technical Disclosure Bulletin, April 1988.
- [WH04] Neil H. E. Weste and David Harris. *CMOS VLSI Design: A Circuits and System Perspective*. Addison-Wesley, 2004.
- [Yeh93] Tse-Yu Yeh. *Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors*. PhD thesis, University of Michigan, Department of Electrical Engineering and Computer Science, 1993.